BLM6112
Advanced Computer Architecture
Data-Level Parallelism in Vector, SIMD, and GPU Architectures - 1

Prof. Dr. Nizamettin AYDIN

naydin@yildiz.edu.tr

http://www3.yildiz.edu.tr/~naydin

Introduction

- A summary of the five mainstream computing classes and their system characteristics:

Feature	Personal mobile device (PMD)	Desktop	Server	Clusters/warehouse-scale computer	Internet of things/ embedded
Price of system	$100–$1000	$300–$2500	$5000–$10,000,000	$100,000–$200,000,000	$10–$100,000
Price of microprocessor	$10–$100	$50–$500	$200–$2000	$50–$250	$0.01–$100
Critical system design issues	Cost, energy, media performance, responsiveness	Price-performance, energy, graphics performance	Throughput, availability, scalability, energy	Price-performance, throughput, energy proportionality	Price, energy, application-specific performance

Introduction

- Classes of Parallelism :
  - Data-Level Parallelism (DLP) arises
    - because there are many data items that can be operated on at the same time.
  - Task-Level Parallelism (TLP) arises
    - because tasks of work are created that can operate independently and largely in parallel.

- Flynn, M. J. (1966). Very high-speed computing systems. Proceedings of the IEEE, 54(12), 1901–1909. doi:10.1109/proc.1966.5273

- Computer hardware in turn can exploit these two kinds of application parallelism in four major ways:

Introduction

- Instruction-level parallelism exploits
  - DLP at modest levels with compiler help using ideas like pipelining and at medium levels using ideas like speculative execution.
- Vector architectures, Graphic Processor Units (GPUs), and multimedia instruction sets exploit
  - DLP by applying a single instruction to a collection of data in parallel.
- Thread-level parallelism exploits either
  - DLP or TLP in a tightly coupled hardware model that allows for interaction between parallel threads.
- Request-level parallelism exploits
  - parallelism among largely decoupled tasks specified by the programmer or the operating system.

Introduction

- Classes of computers (interms of # of processors):
  - Single Instruction stream, Single Data stream (SISD)
    - This category is the uniprocessor.
    - The programmer thinks of it as the standard sequential computer, but it can exploit ILP.
  - Single Instruction stream, Multiple Data streams (SIMD)
    - The same instruction is executed by multiple processors using different data streams.
    - SIMD computers exploit DLP by applying the same operations to multiple items of data in parallel.
    - Each processor has its own data memory (hence, the MD of SIMD), but there is a single instruction memory and control processor, which fetches and dispatches instructions.

Introduction

- Multiple Instruction streams, Single Data stream (MISD)
  - No commercial multiprocessor of this type has been built to date, but it rounds out this simple classification.
- Multiple Instruction streams, Multiple Data streams (MIMD)
  - Each processor fetches its own instructions and operates on its own data, and it targets TLP.
  - In general, MIMD is more flexible than SIMD and thus more generally applicable, but it is inherently more expensive than SIMD.

SIMD Parallelism

- A question for the SIMD architecture has always been just how wide a set of applications has significant DLP.
- SIMD architectures can exploit significant DLP for:
  - Matrix-oriented scientific computing
  - Media-oriented image and sound processing
  - Machine learning algorithms

SIMD Parallelism

- SIMD is more energy efficient than Multiple Instruction Multiple Data (MIMD)
  - MIMD needs to fetch one instruction per data operation
  - In SIMD, a single instruction can launch many data operations
  - Makes SIMD attractive for personal mobile devices as well as for servers
    - SIMD allows programmer to continue to think sequentially yet achieves parallel speedup by having parallel data operations

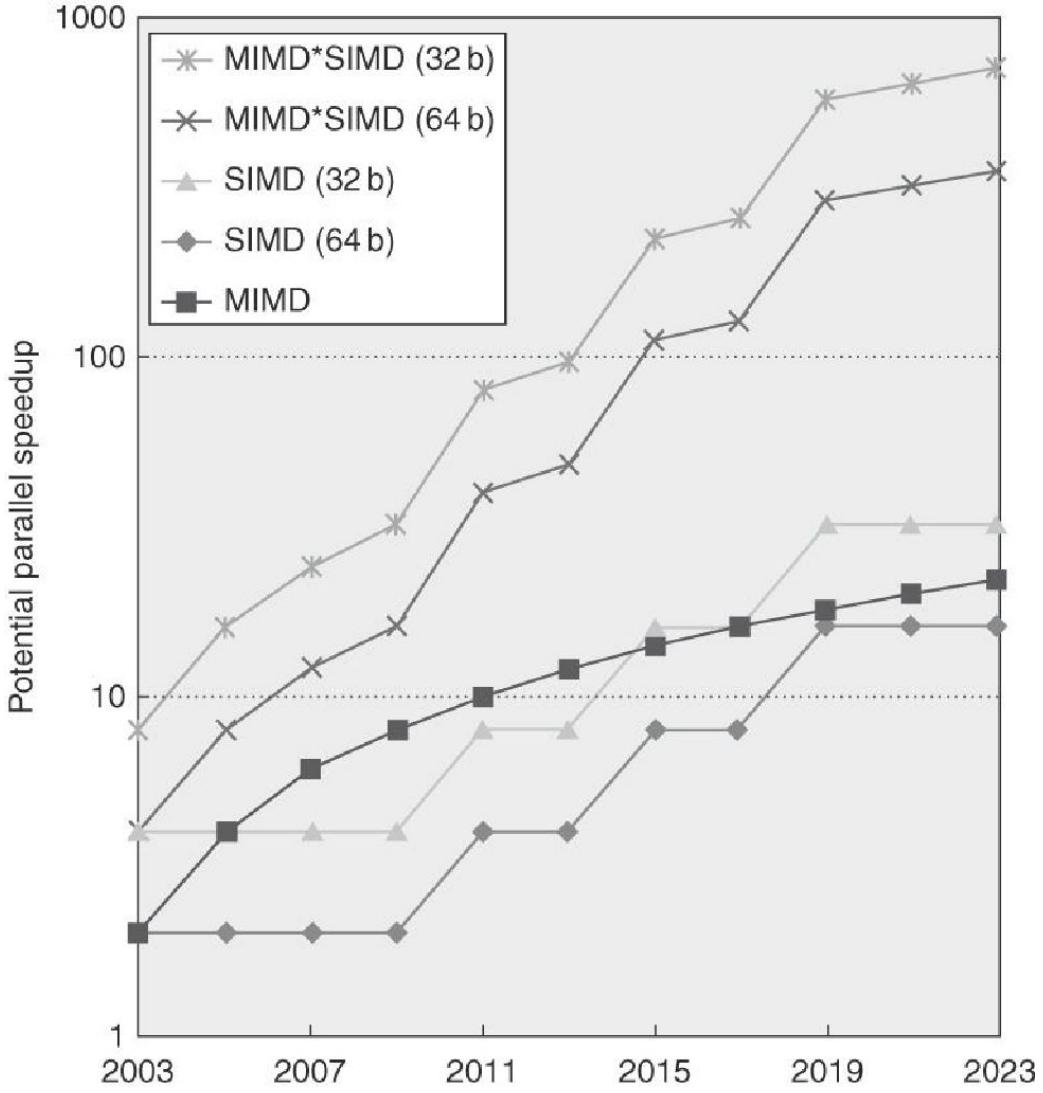
SIMD Parallelism

- There are three variations of SIMD
  - Vector architectures
    - Extends pipelined execution of many data operations.
    - Easier to understand and to compile, but they were considered too expensive for microprocessors until recently.
      - Part of that expense was in transistors, and part was in the cost of sufficient dynamic random access memory (DRAM) bandwidth, given the widespread reliance on caches to meet memory performance demands on conventional microprocessors.
  - Multimedia SIMD instruction set extensions
    - found in most instruction set architectures that support multimedia applications.
      - For x86 architectures, the SIMD instruction extensions started with the MMX (multimedia extensions) in 1996, which were followed by several SSE (streaming SIMD extensions) versions in the next decade, and they continue until this day with AVX (advanced vector extensions).
      - To get the highest computation rate from an x86 computer, you often need to use these SIMD instructions, especially for floating-point programs.

SIMD Parallelism

- Graphics Processing Units (GPUs)
  - comes from the graphics accelerator community, offering higher potential performance than is found in traditional multicore computers today.
  - Although GPUs share features with vector architectures, they have their own distinguishing characteristics, in part because of the ecosystem in which they evolved.
  - This environment has a system processor and system memory in addition to the GPU and its graphics memory. In fact, to recognize those distinctions, the GPU community refers to this type of architecture as heterogeneous.

SIMD vs. MIMD

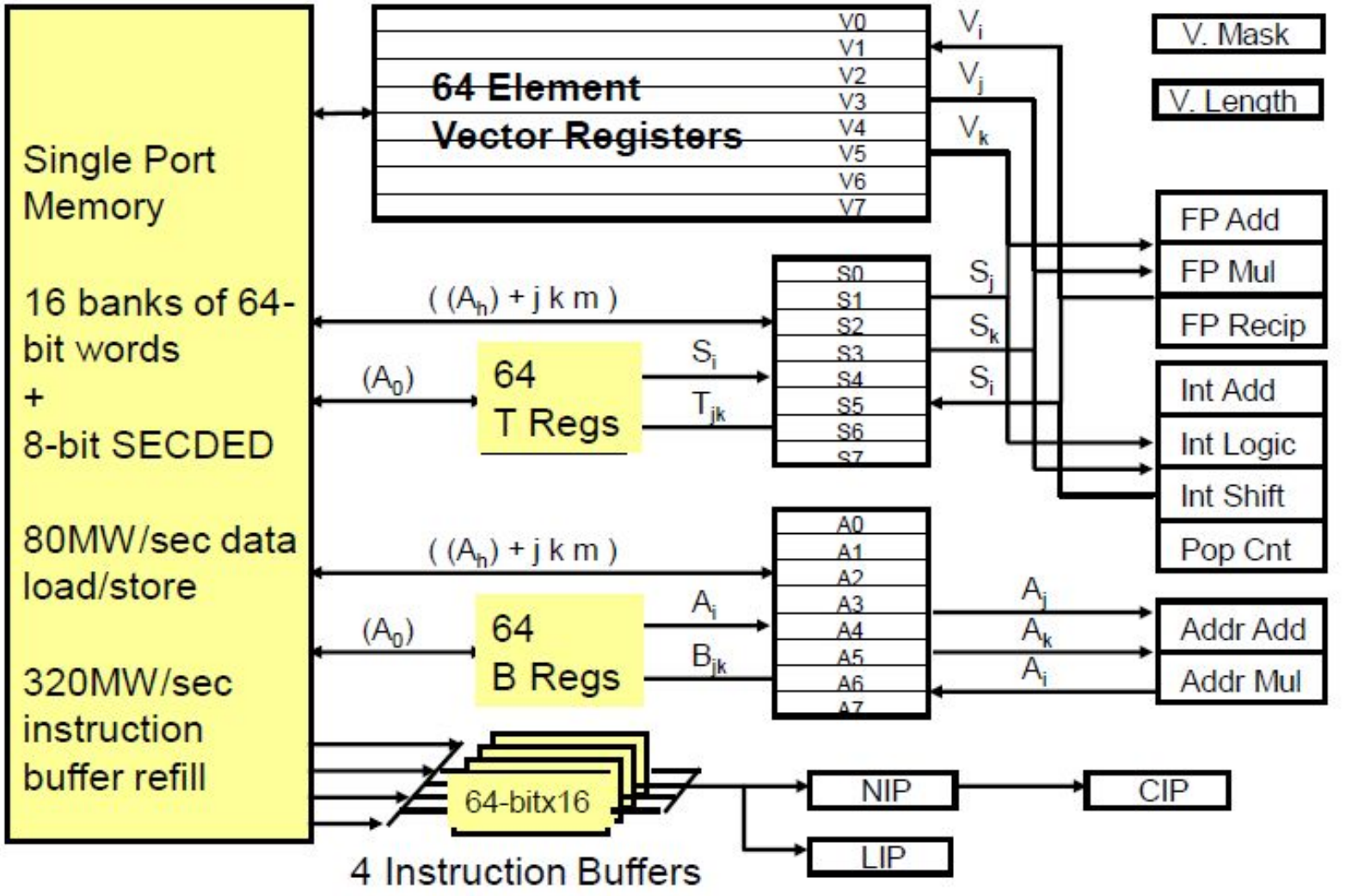


- For x86 processors:
  - Expect two additional cores per chip per year
  - SIMD width to double every four years
  - Potential speedup from SIMD to be twice that from MIMD!

Vector Supercomputers

- In 70-80s, Supercomputer ≡ Vector machine
- Definition of supercomputer
  - Fastest machine in the world at given task
  - A device to turn a compute-bound problem into an I/O bound problem
  - CDC6600 (Cray, 1964) is regarded as the first supercomputer
- Vector supercomputers (epitomized by Cray 1, 1976)
  - Scalar unit + vector extensions
    - Vector registers, vector instructions
    - Vector loads/stores
    - Highly pipelined functional units

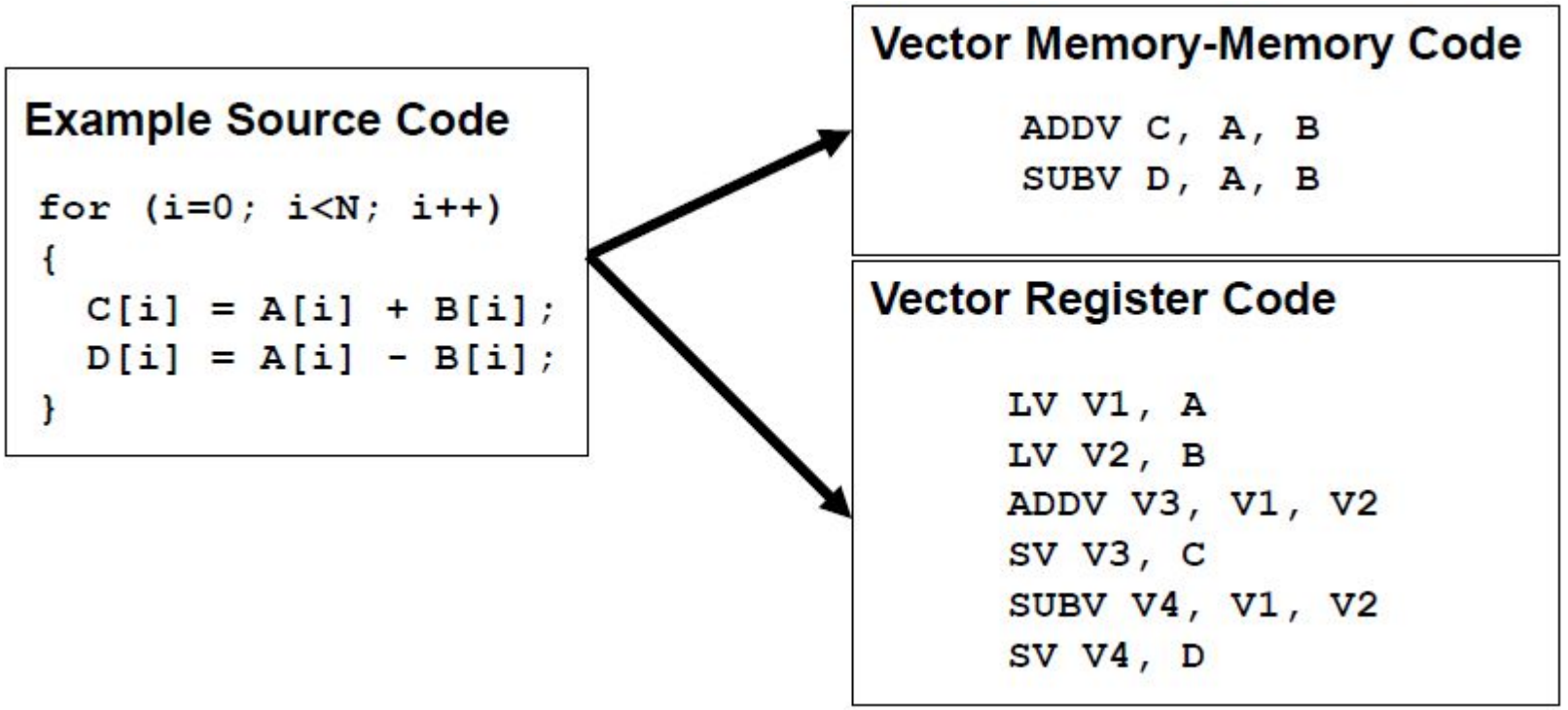
Cray-1 (1976)



memory bank cycle 50 ns *processor cycle* 12.5 ns (80MHz)

Vector Memory Memory vs. Vector Register Machines

- Vector memory-memory instructions hold all vector operands in main memory
  - The first vector machines, CDC Star 100 (1973) and TI ASC (1971), were memory-memory machines
- Cray 1 (1976) was first vector register machine



Vector Memory Memory vs. Vector Register Machines

- Vector memory-memory architectures (VMMA) require greater main memory bandwidth, why?
  - All operands must be read in and out of memory
- VMMAs make it difficult to overlap execution of multiple vector operations, why?
  - Must check dependencies on memory addresses
- VMMAs incur greater startup latency
  - Scalar code was faster on CDC Star-100 for vectors < 100 elements
  - For Cray-1, vector/scalar breakeven point was around 2 elements
    - Apart from CDC follow ons (Cyber-205, ETA-10) all major vector machines since Cray-1 have had vector register architectures

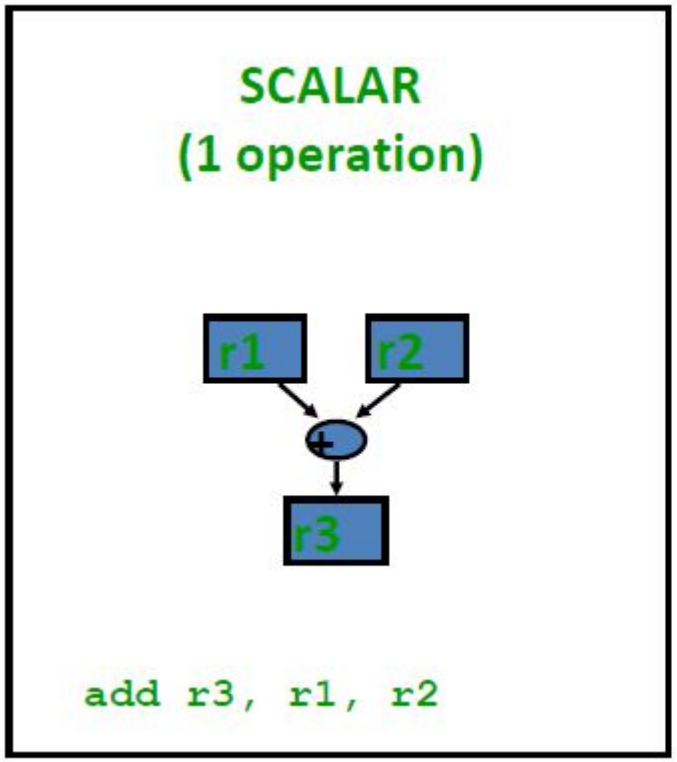
Vector Architectures

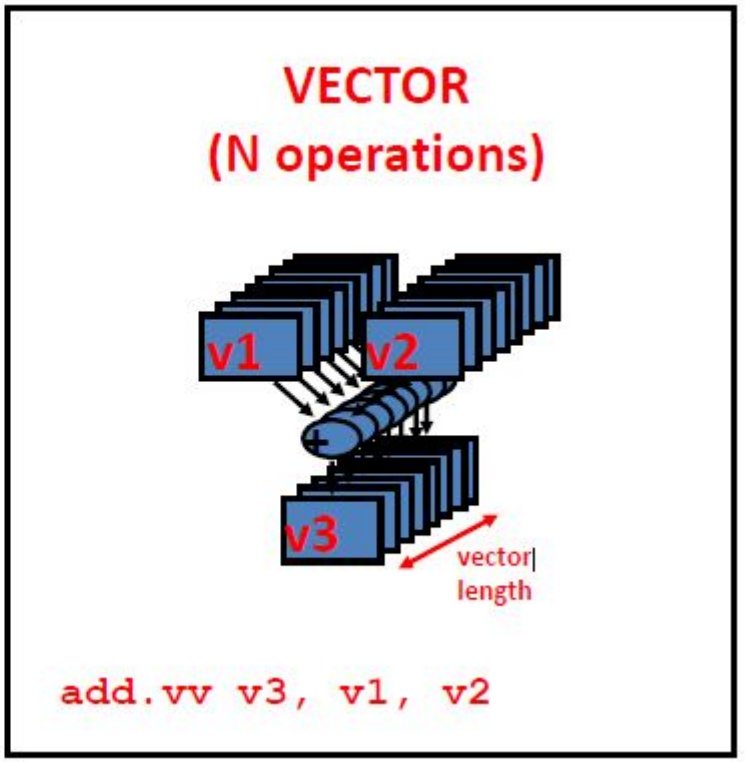
- Basic idea:
  - Read sets of data elements into vector registers
  - Operate on data in those register files
  - Disperse the results back into memory
- Vector loads/stores are deeply pipelined
  - Program pays the long memory latency only once per vector load/store vs. latency for each element for regular load/store.
- Register files are controlled by compiler
  - Register files act as compiler controlled buffers
  - Used to hide memory latency
  - Leverage memory bandwidth

- Kozyrakis, C., & Patterson, D. (n.d.). Vector vs. superscalar and VLIW architectures for embedded multimedia benchmarks. 35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002. (MICRO-35). Proceedings. doi:10.1109/micro.2002.1176257

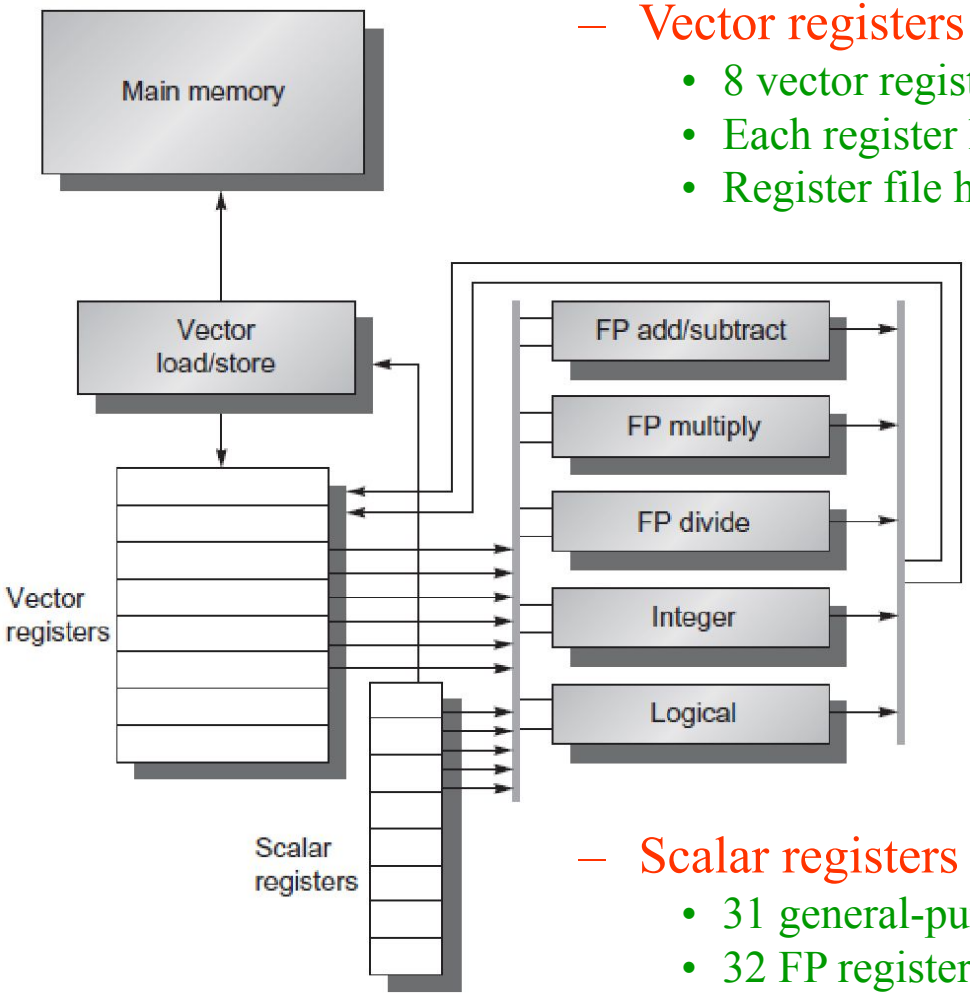
Vector Processing Model

- Vector processors have high-level operations that work on linear arrays of numbers: vectors



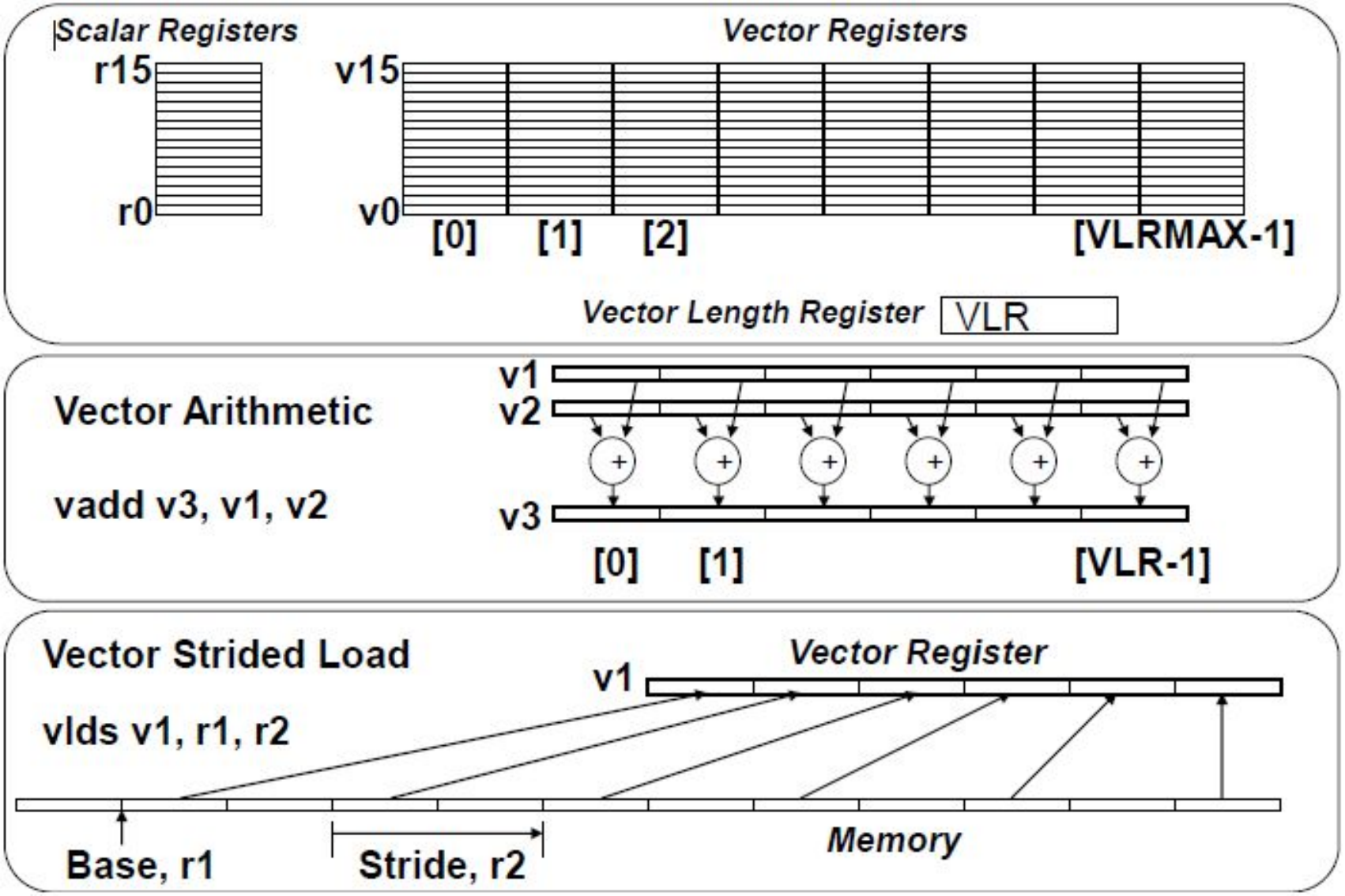


RISC-V Vector (RVV) ISA Example: RV64V



- Loosely based on Cray-1
  - Vector registers
    - 8 vector registers in this example
    - Each register holds a 32-element, 64 bits/element vector
    - Register file has 16 read ports and 8 writeports
  - Vector functional units
    - 5 FUs in this example
    - Fully pipelined
    - Data and control hazards are detected
  - Vector load-store unit
    - Fully pipelined
    - Words move between registers and memory
    - One word per clock cycle after initial latency
  - Scalar registers
    - 31 general-purpose registers
    - 32 FP registers point registers (RV64V)

Vector Programming Model



Vector Instruction Set Advantages

- Compact
  - One short instruction encodes N operations
- Expressive
  - tells hardware that these N operations are independent
  - N operations use the same functional unit
  - N operations access disjoint registers
  - N operations access registers in the same pattern as previous instruction
  - N operations access a contiguous block of memory (unit stride load/store)
  - N operations access memory in a known pattern (stridden load/store)
- Scalable
  - Can run same object code on more parallel pipelines or lanes

The RV64V Instructions

Mnemonic	Name	Description
vadd	ADD	Add elements of V[rs1] and V[rs2], then put each result in V[rd]
vsub	SUBtract	Subtract elements of V[rs2] frpm V[rs1], then put each result in V[rd]
vmul	MULtiply	Multiply elements of V[rs1] and V[rs2], then put each result in V[rd]
vdiv	DIVide	Divide elements of V[rs1] by V[rs2], then put each result in V[rd]
vrem	REMainder	Take remainder of elements of V[rs1] by V[rs2], then put each result in V[rd]
vsqrt	SQuare RooT	Take square root of elements of V[rs1], then put each result in V[rd]
vsll	Shift Left	Shift elements of V[rs1] left by V[rs2], then put each result in V[rd]
vsrl	Shift Right	Shift elements of V[rs1] right by V[rs2], then put each result in V[rd]
vsra	Shift Right Arithmetic	Shift elements of V[rs1] right by V[rs2] while extending sign bit, then put each result in V[rd]
vxor	XOR	Exclusive OR elements of V[rs1] and V[rs2], then put each result in V[rd]
vor	OR	Inclusive OR elements of V[rs1] and V[rs2], then put each result in V[rd]
vand	AND	Logical AND elements of V[rs1] and V[rs2], then put each result in V[rd]
vsgnj	SiGN source	Replace sign bits of V[rs1] with sign bits of V[rs2], then put each result in V[rd]
vsgnjn	Negative SiGN source	Replace sign bits of V[rs1] with complemented sign bits of V[rs2], then put each result in V[rd]
vsgnjx	Xor SiGN source	Replace sign bits of V[rs1] with xor of sign bits of V[rs1] and V[rs2], then put each result in V[rd]

The RV64V Instructions

vld	Load	Load vector register V[rd] from memory starting at address R[rs1]
vlds	Strided Load	Load V[rd] from address at R[rs1] with stride in R[rs2] (i.e., R[rs1]+i×R[rs2])
vldx	Indexed Load (Gather)	Load V[rs1] with vector whose elements are at R[rs2]+V[rs2] (i.e., V[rs2] is an index)
vst	Store	Store vector register V[rd] into memory starting at address R[rs1]
vsts	Strided Store	Store V[rd] into memory at address R[rs1] with stride in R[rs2] (i.e., R[rs1]+i×R[rs2])
vstx	Indexed Store (Scatter)	Store V[rs1] into memory vector whose elements are at R[rs2]+V[rs2] (i.e., V[rs2] is an index)
vpeq	Compare =	Compare elements of V[rs1] and V[rs2]. When equal, put a 1 in the corresponding 1-bit element of p[rd]; otherwise, put 0
vpne	Compare !=	Compare elements of V[rs1] and V[rs2]. When not equal, put a 1 in the corresponding 1-bit element of p[rd]; otherwise, put 0
vplt	Compare <	Compare elements of V[rs1] and V[rs2]. When less than, put a 1 in the corresponding 1-bit element of p[rd]; otherwise, put 0
vpxor	Predicate XOR	Exclusive OR 1-bit elements of p[rs1] and p[rs2], then put each result in p[rd]
vpor	Predicate OR	Inclusive OR 1-bit elements of p[rs1] and p[rs2], then put each result in p[rd]
vpand	Predicate AND	Logical AND 1-bit elements of p[rs1] and p[rs2], then put each result in p[rd]
setvl	Set Vector Length	Set vl and the destination register to the smaller of mvl and the source regsiter

The RV64V vector instructions

- All use the R instruction format.
- Each vector operation with two operands is shown with both operands being vector (.vv)
- There are also versions where
  - the second operand is a scalar register (.vs)
  - the first operand is a scalar register and the second is a vector register (.sv).

The RV64V vector instructions

- Operate on many elements concurrently
  - Allows use of slow but wide execution units
    - High performance, lower power
- Independence of elements within a vector instruction
  - Allows scaling of functional units without costly dependencechecks
- Flexible
  - 32 64-bit / 128 16-bit / 256 8-bit
  - Matches the need of multimedia (8bit), scientific applications that require high precision

How Vector Processors Work: An Example

- We can best understand a vector processor by looking at a vector loop for RV64V.
- Let's take a typical vector problem

  $Y = a \times X + Y$

  – X and Y are vectors, initially resident in memory, and a is a scalar.
    - This problem is the SAXPY (single-precision a X plus Y) or DAXPY (double precision a X plus Y) loop that forms the inner loop of the Linpack benchmark
      – (Dongarra et al., 2003, https://doi.org/10.1002/cpe.728).
    - Linpack is a collection of linear algebra routines, and the Linpack benchmark consists of routines for performing Gaussian elimination.

RV64V Example: DAXPY Loop

- Show the code for RV64G and RV64V for the DAXPY loop.
  - for (i=0; i<32; i++)
    Y[i] = a × X[i] + Y[i]
    - adds a scalar multiple of a double precision vector to another double precision vector
- For this example, assume that
  - X and Y have 32 elements
  - the number of elements (i.e. Length) of the vectors matches the length of the vector operation.
  - the starting addresses of X and Y are in x5 and x6

Answer

- Here is the RISC-V code:

```
        fld     f0,a              # Load scalar a
        addi    x28 , x5, #256    # Last address to load
Loop:   fld     f1 , 0(x5)        # Load X[i]
        fmul.d  f1 , f1 , f0      # a × X[i]
        fld     f2 , 0(x6)        # Load Y[i]
        fadd.d  f2 , f2 , f1      # a  X[i] + Y[i]
        fsd     f2 , 0(x6)        # Store into Y[i]
        addi    x5 ,x5 , #8       # Increment index to X
        addi    x6 , x6 , #8      # Increment index to Y
        bne     x28 , x5 , Loop   # Check if done
```

Answer

- Here is the RV64V code for DAXPY:

```
vsetdcfg 4*FP64       # Enable 4 DP FP vector registers
fld      f0 , a       # Load scalar a
vld      v0 , x5      # Load vector X
vmul     v1 , v0 , f0 # Vector-scalar multiply
vld      v2 , x6      # Load vector Y
vadd     v3 , v1 , v2 # Vector-vector add
vst      v3 , x6      # Store the sum
vdisable              # Disable vector registers
```

  - Note that the assembler determines which version of the vector operations to generate.
    - Because the multiply has a scalar operand, it generates vmul.vs, whereas the add doesn't, so it generates vadd.vv.
  - The initial instruction configures the first four vector registers to hold 64-bit floating-point data.
  - The last instruction disables all vector registers.

Answer

RV64G Code

```
      fld      f0,a
      addi     x28 , x5, #256
Loop: fld      f1 , 0(x5)
      fmul.d   f1 , f1 , f0
      fld      f2 , 0(x6)
      fadd.d   f2 , f2 , f1
      fsd      f2 , 0(x6)
      addi     x5 ,x5 , #8
      addi     x6 , x6 , #8
      bne      x28 , x5 , Loop
```

RV64V Code

```
vsetdcfg   4*FP64
fld        f0 , a
vld        v0 , x5
vmul       v1 , v0 , f0
vld        v2 , x6
vadd       v3 , v1 , v2
vst        v3 , x6
vdisable
```

Answer

- 8 RV64V vector instructions vs. 258 RV64G scalar instructions
- In RV64G Code
  - Fadd.d must wait for fmul.d
  - fsd must wait for fadd.d
  - Lots of pipeline stalls are necessary for deeply pipelined architecture.
- In RV64V Code
  - Stall once for the first vector element, subsequent elements will flow smoothly down the pipeline.
  - Pipeline stalls are required only once per vector instruction, rather than once per vector element
- Pipeline stall frequency on RV64G will be about 32× higher than it is on RV64V.

Remarks

- The most dramatic difference between the scalar and vector code is that the vector processor greatly reduces the dynamic instruction bandwidth, executing only 8 instructions versus 258 for RV64G.
- When the compiler produces vector instructions for such a sequence, and the resulting code spends much of its time running in vector mode,
  - the code is said to be vectorized or vectorizable.
- Loops can be vectorized when they do not have dependences between iterations of a loop,
  - which are called loop-carried dependences

Example

- A common use of multiply-accumulate operations is to multiply using narrow data and to accumulate at a wider size to increase the accuracy of a sum of products.

- Show how the preceding code would change if X and a were single-precision instead of a double-precision floating point.
- Next, show the changes to this code if we switch X, Y, and a from floating-point type to integers.

Answer

- The same code works with two small changes:
  - The configuration instruction includes one single-precision vector,
  - the scalar load is now single-precision:

```
vsetdcfg      1*FP32 , 3*FP64  # 1 32b, 3 64b vregs
flw       f0 , a           # Load scalar a
vld       v0 , x5          # Load vector X
vmul          v1 , v0 , f0         # Vector-scalar mult
vld       v2 , x6          # Load vector Y
vadd          v3 , v1 , v2         # Vector-vector add
vst       v3 , x6          # Store the sum
vdisable                   # Disable vector regs
```

Answer

- RV64V hardware will implicitly perform a conversion from the single-precision to the double-precision in this setup.
- We must use an integer load instruction and integer register to hold the scalar value:

```
vsetdcfg      1*X32,3*X64      # 1 32b, 3 64b int reg
lw        x7 , a           # Load scalar a
vld       v0 , x5          # Load vector X
vmul          v1 , v0 , x7         # Vector-scalar mult
vld       v2 , x6          # Load vector Y
vadd          v3 , v1 , v2         # Vector-vector add
vst       v3 , x6          # Store the sum
vdisable                   # Disable vector regs
```

Challenges of Vector Instructions

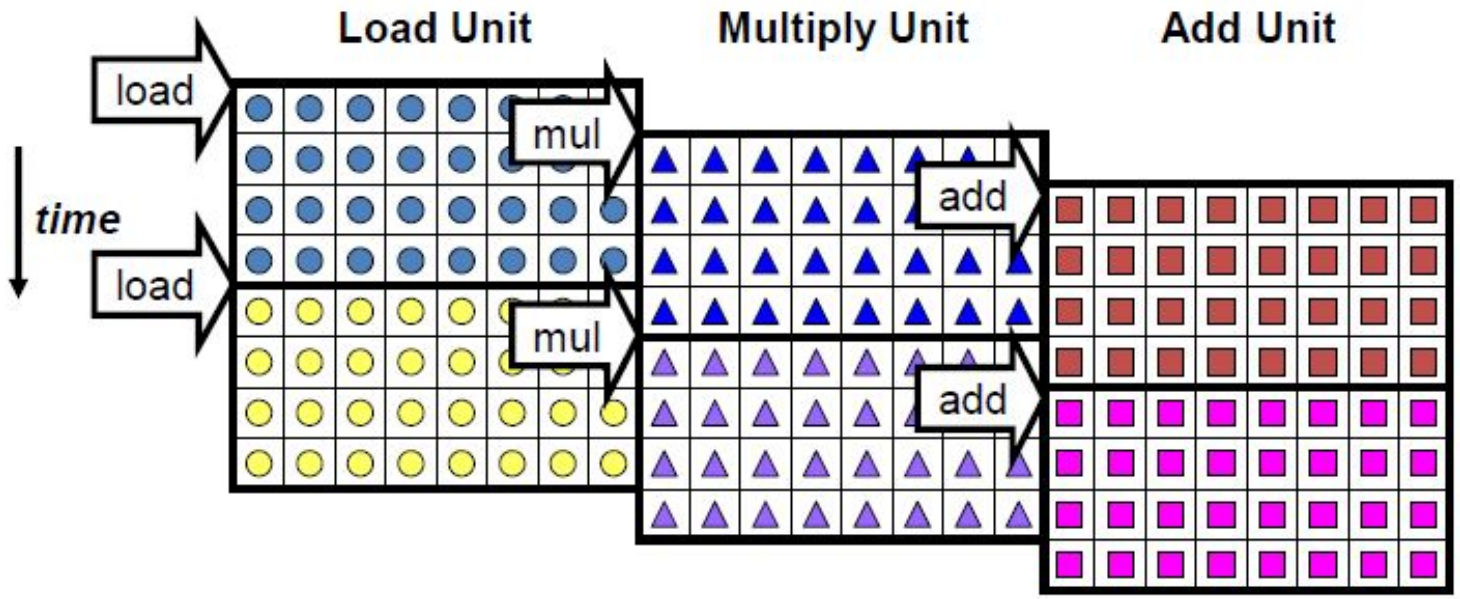
- Start up time
  - Application and architecture must support long vectors.
    - Otherwise, they will run out of instructions requiring ILP
- Long latency of vector functional unit
  - Assume the same as Cray-1
    - Floating point add => 6 clock cycles
    - Floating point multiply => 7 clock cycles
    - Floating point divide => 20 clock cycles
    - Vector load => 12 clock cycles

Vector Execution Time

- Execution time of a sequence of vector operations depends on three factors:
  - Length of operand vectors
  - Structural hazards among the operations
  - Data dependencies
- Modern vector computers have vector functional units with multiple parallel pipelines that can produce two or more results per clock cycle
- RV64V functional units consume one element per clock cycle for individual operations
  - Thus the execution time in clock cycles for a single vector instruction is approximately the vector length
- Efficient way to estimate the execution time:
  - Convoy and chime

Vector Instruction Parallelism

- Can overlap execution of multiple vector instructions
  - example machine has 32 elements per vector register and 8 lanes
  - Complete 24 operations/cycle while issuing 1 short instruction/cycle



Convoy

- Convoy
  - Set of vector instructions that could potentially execute together
- Instructions in a convoy must not contain any structural hazards;
  - if such hazards were present, the instructions would need to be serialized and initiated in different convoys.
    - Thus the vld and the following vmul in the preceding example can be in the same convoy.
- One can estimate performance of a section of code by counting the number of convoys.
- It is assumed that a convoy of instructions must complete execution before any other instructions (scalar or vector) can begin execution.

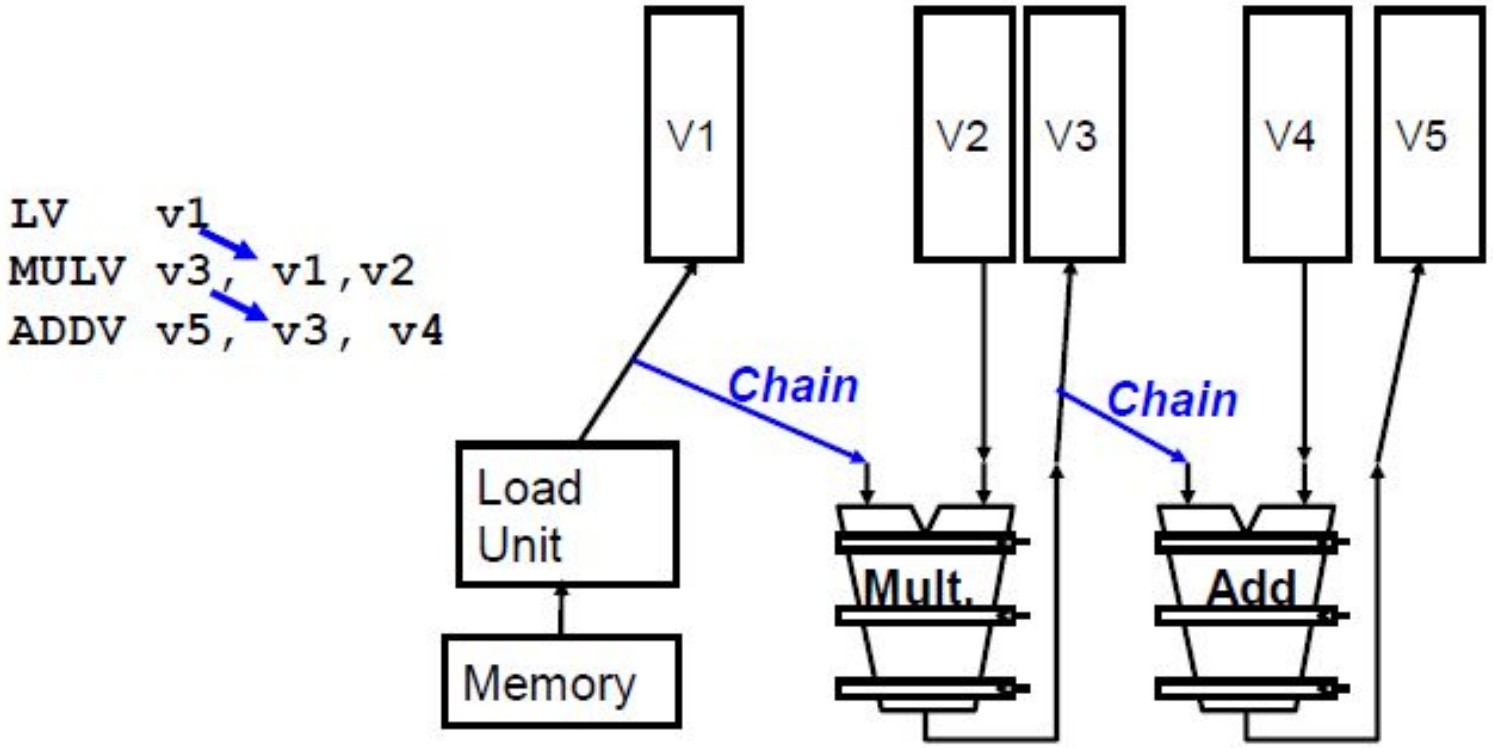
Vector Chaining

- Sequences with read-after-write (RAW) dependency hazards should be in separate convoy.
  - However, chaining allows them to be in the same convoy
- Chaining
  - Allows a vector operation to start as soon as the individual elements of its vector source operand become available
    - the results from the first functional unit in the chain are forwarded to the second functional unit
- Chaining is implemented by allowing the processor to read and write a particular vector register at the same time

- Recent implementations use flexible chaining, which allows a vector instruction to chain to essentially any other active vector instruction, assuming that we don't generate a structural hazard.
- All modern vector architectures support flexible chaining

Vector Chaining

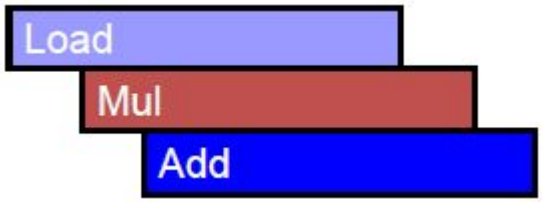
- Vector version of register bypassing
  - Allows a vector operation to start as soon as the individual elements of its vector source operand become available



Advantages of Vector Chaining

- Without chaining,
  - must wait for last element of result to be written before starting dependent instruction

- With chaining,
  - can start dependent instruction as soon as first result appears



Chimes

- To turn convoys into execution time, we need a metric to estimate the length of a convoy.
- Chime
  - Unit of time taken to execute one convoy
- a vector sequence that consists of *m* convoys executes in *m* chimes;
  - for a vector length of *n*, for RV64V implementation, this is approximately $m \times n$ clock cycles.
- Chime approximation ignores some processor-specific overheads, many of which are dependent on vector length.
  - Therefore measuring time in chimes is a better approximation for long vectors than for short ones.

Execution Time Example

- Show how the following code sequence lays out in convoys, assuming a single copy of each vector functional unit:

```
vld  v0 , x5          # Load vector X
vmul    v1 , v0 , f0        # Vector-scalar multiply
vld  v2 , x6          # Load vector Y
vadd    v3 , v1 , v2        # Vector-vector add
vst  v3 , x6          # Store the sum
```

  - How many chimes will this vector sequence take?
  - How many cycles per FLOP (floating-point operation) are needed, ignoring vector instruction issue overhead?

Answer

- 3 Convoys:
  - 1st convoy starts with the 1st vld instruction.
    - vmul is dependent on the 1st vld,
      - but chaining allows it to be in the same convoy.
  - 2nd vld instruction must be in a separate convoy because there is a structural hazard on the load/store unit for the prior vld instruction.
    - vadd is dependent on the 2nd vld,
      - but it can be in the same convoy via chaining.
  - vst has a structural hazard on the vld in the 2nd convoy,
    - so it must go in the third convoy.

Answer

- This analysis leads to the following layout of vector instructions into convoys:

1	vld	vmul
2	vld	vadd
3	vst	

- The sequence requires 3 convoys.
- Because the sequence takes 3 chimes and there are 2 fp operations per result, the number of cycles per FLOP is 1.5
- This example shows that the chime approximation is reasonably accurate for long vectors.
  - For example, for 32-element vectors, the time in chimes is 3, so the sequence would take about 32×3 or 96 clock cycles.

Challenges

- Most important source of overhead ignored by the chime model is vector start-up time,
  - which is the latency in clock cycles until the pipeline is full.
- Start-up time is principally determined by the pipelining latency of the vector functional unit.
  - For RV64V, same pipeline depths as the Cray-1 will be assumed.
    - All functional units are fully pipelined.
    - Pipeline depths are
      - 6 clock cycles for fp add,
      - 7 for fp multiply,
      - 20 for fp divide,
      - 12 for vector load.

Improvements

- Optimizations that either improve the performance or increase the types of programs that can run well on vector architectures:
- How can a vector processor execute a single vector faster than one element per clock cycle?
  - Multiple elements per clock cycle improve performance.
- How does a vector processor handle programs where the vector lengths are not the same as the maximum vector length (mvl)?
  - Because most application vectors don't match the architecture vector length, we need an efficient solution to this common case.

Improvements

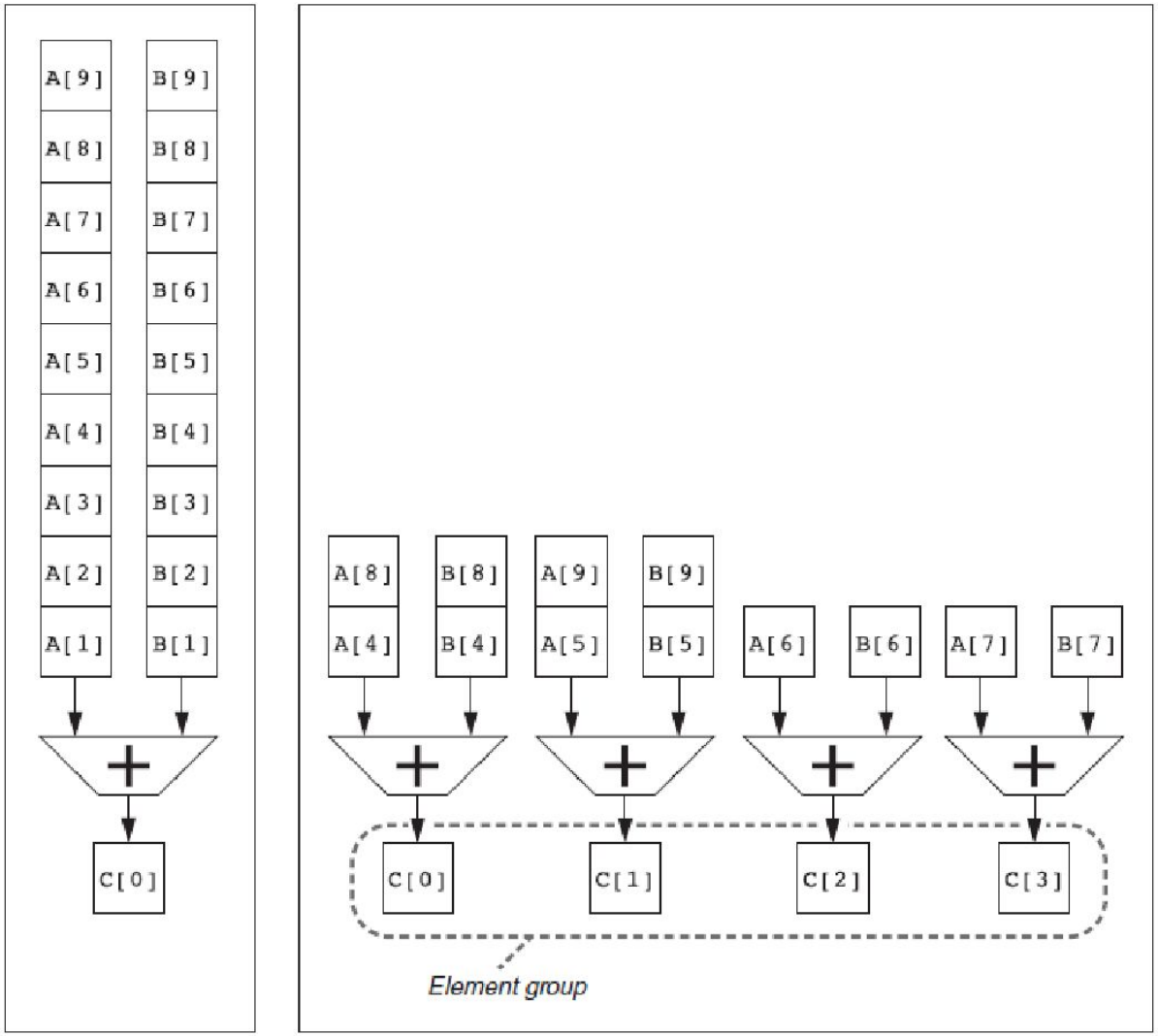
- What happens when there is an IF statement inside the code to be vectorized?
  - More code can vectorize if we can efficiently handle conditional statements.
- What does a vector processor need from the memory system?
  - Without sufficient memory bandwidth, vector execution can be futile.
- How does a vector processor handle multiple dimensional matrices?
  - This popular data structure must vectorize for vector architectures to do well.
- How does a vector processor handle sparse matrices?
  - This popular data structure must vectorize also.
- How do you program a vector computer?
  - Architectural innovations that are a mismatch to programming languages and their compilers may not get widespread use.

Multiple Lanes

- A critical advantage of a vector instruction set
  - allows software to pass a large amount of parallel work to hardware using only a single short instruction.
    - One vector instruction can include scores of independent operations yet be encoded in the same number of bits as a conventional scalar instruction.
- The parallel semantics of a vector instruction allow an implementation to execute these elemental operations using a deeply pipelined functional unit,
  - an array of parallel functional units; or a combination of parallel and pipelined functional units.
- Next figure illustrates how to improve vector performance by using parallel pipelines to execute a vector add instruction.

Multiple Lanes

- Using multiple functional units to improve the performance of a single vector add instruction, C=A+B.

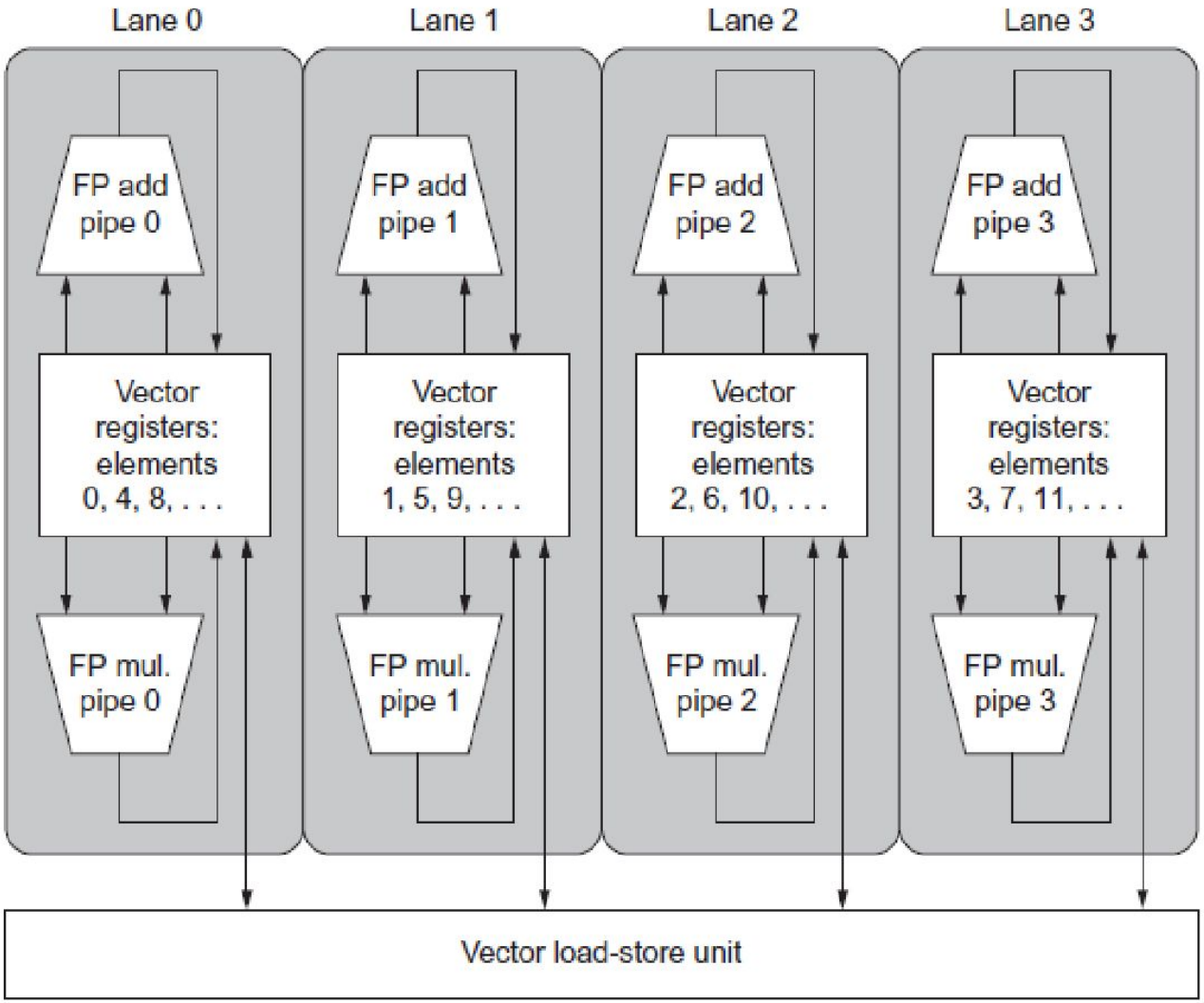


- The vector processor (A) on the left has a single add pipeline and can complete one addition per clock cycle.
- The vector processor (B) on the right has four add pipelines and can complete four additions per clock cycle.
- The elements within a single vector add instruction are interleaved across the four pipelines.
- The set of elements that move through the pipelines together is termed an element group.

Multiple Lanes

- In RV64V instruction set, all vector arithmetic instructions only allow element *N* of one vector register to take part in operations with element *N* from other vector registers.
  - This dramatically simplifies the design of a highly parallel vector unit, which can be structured as multiple parallel lanes.
  - As with a traffic highway, we can increase the peak throughput of a vector unit by adding more lanes.
- Next figure shows the structure of a four-lane vector unit.
  - Thus going to four lanes from one lane reduces the number of clocks for a chime from 32 to 8.

Multiple Lanes



- Vector register memory is divided across the lanes, with each lane holding every fourth element of each vector register.
- Three vector functional units:
  - an FP add,
  - an FP multiply,
  - a load-store unit.
- Each of the vector arithmetic units contains four execution pipelines, one per lane, which act in concert to complete a single vector instruction.

Vector-Length Registers:
Handling Loops Not Equal to 32

- A vector register processor has a natural vector length determined by the maximum vector length (mvl) (32 in the example above).
- In a real program, the length of a particular vector operation is unknown at compile time.
  - In fact, a single piece of code may require different vector lengths.
- For example, consider the following code:

```
for (i = 0;  i < n; i = i + 1)
    Y[i] = a * X[i] + Y[i];
```

Vector-Length Registers

- Solution to these problems is to add a vector-length register (vl).
  - The vl controls the length of any vector operation, including a vector load or store.
    - The value in the vl cannot be greater than the mvl.
- This solves the problem as long as the real length is less than or equal to the maximum vector length (mvl).
- This parameter means the length of vector registers can grow in later computer generations without changing the instruction set.

Vector Length Register

- RV64V code for vector DAXPY for any value of *n*.

```
    vsetdcfg    2 DP FP      # Enable 2 64b Fl.Pt. registers
    fld         f0 , a       # Load scalar a
loop: setvl       t0 ,a0          # vl = t0 = min(mvl,n)
    vld         v0 , x5      # Load vector X
    slli        t1 , t0 , 3  # t1 = vl * 8 (in bytes)
    add         x5 , x 5, t1 # Increment pointer to X by vl*8
    vmul        v0 , v0 , f0 # Vector-scalar mult
    vld         v1 , x6      # Load vector Y
    vadd        v1 , v0 , v1 # Vector-vector add
    sub         a0 , a0 , t0 # n -= vl (t0)
    vst         v1 , x6      # Store the sum into Y
    add         x6 , x6 , t1 # Increment pointer to Y by vl*8
    bnez        a0 , loop    # Repeat if n != 0
    vdisable              # Disable vector regs}
```

Predicate Registers:
Handling IF Statements in Vector Loops

- Main reasons for lower levels of vectorization:
  - presence of conditionals (IF statements) inside loops
  - use of sparse matrices
- Programs that contain IF statements in loops cannot be run in vector mode because
  - IF statements introduce control dependences into a loop.
- Consider the following loop written in C:

```
for (i = 0; i < 64;   i = i + 1)
    if (X[i] != 0)
        X[i] = X[i] – Y[i];
```

  - This loop cannot normally be vectorized because of the conditional execution of the body

Predicate Registers:
Handling IF Statements in Vector Loops

- However, if the inner loop could be run for the iterations for which X[i] ≠ 0, then the subtraction could be vectorized.
  - The common extension for this capability is vector-mask control.
    - In RV64V, predicate registers hold the mask and essentially provide conditional execution of each element operation in a vector instruction.
- Predicate registers are configured and can be disabled.
  - Enabling a predicate register initializes it to all 1 s,
    - meaning that subsequent vector instructions operate on all vector elements.
    - Following code can be used for the previous loop, assuming that the starting addresses of X and Y are in x5 and x6, respectively:

```
vsetdcfg   2*FP64          # Enable 2 64b FP vector regs
vsetpcfgi  1           # Enable 1 predicate register
vld        v0 , x5         # Load vector X into v0
vld        v1 , x6         # Load vector Y into v1
fmv.d.x    f0 , x0         # Put (FP) zero into f0
vpne       p0 , v0 , f0    # Set p0(i) to 1 if v0(i)!=f0
vsub       v0 , v0 , v1# Subtract under vector mask
vst        v0 , x5         # Store the result in X
vdisable               # Disable vector registers
vpdisable              # Disable predicate registers
```

Predicate Registers

- Using a vector-mask register does have overhead.
  - With scalar architectures, conditionally executed instructions still require execution time when the condition is not satisfied.
- Elimination of a branch and the associated control dependences can make a conditional instruction faster even if it sometimes does useless work.
- Vector instructions executed with a vector mask still take the same execution time, even for the elements where the mask is zero.
  - Despite a significant number of zeros in the mask, using vector-mask control may still be significantly faster than using scalar mode.

Memory Banks:
Supplying Bandwidth for Vector Load/Store Units

- Behavior of load/store vector unit is significantly more complicated than that of the arithmetic functional units.
- Start-up time for a load is the time to get the first word from memory into a register.
  - If the rest of the vector can be supplied without stalling, then the vector initiation rate is equal to the rate at which new words are fetched or stored.
- Unlike simpler functional units, the initiation rate may not necessarily be 1 clock cycle because memory bank stalls can reduce effective throughput.

Memory Banks

- Memory system must be designed to support high bandwidth for vector loads and stores
- Spreading accesses across multiple independent memory banks usually delivers the desired rate
- To maintain an initiation rate of one word fetched or stored per clock cycle, the memory system must be capable of producing or accepting this much data.
- Having significant numbers of banks is useful for dealing with vector loads or stores that access rows or columns of data.

Memory Banks

- Most vector processors use memory banks, which allow several independent accesses rather than simple memory interleaving for three reasons:
  - Many vector computers support many loads or stores per clock cycle, and the memory bank cycle time is usually several times larger than the processor cycle time.
    - To support simultaneous accesses from multiple loads or stores, the memory system needs multiple banks and needs to be able to control the addresses to the banks independently.
  - Most vector processors support the ability to load or store data words that are not sequential.
    - In such cases, independent bank addressing, rather than interleaving, is required.
  - Most vector computers support multiple processors sharing the same memory system, so each processor will be generating its own separate stream of addresses.
- In combination, these features lead to the desire for a large number of independent memory banks, as the following example shows.

Example (Cray T90)

- Cray T932 has 32 processors, each capable of generating 4 loads and 2 stores per clock cycle.
  - Processor clock cycle is 2.167 ns,
  - Cycle time of the SRAMs used for the memory system is 15 ns.
- Calculate the minimum number of memory banks required to allow all processors to run at the full memory bandwidth.
- **Answer**
  - The maximum number of memory references each cycle:
    - 32 processors × 6 references per processor = 192
  - Each SRAM bank is busy for 15/2.167 = 6.92 clock cycles,
    - which is rounded up to 7 processor clock cycles.
  - Therefore we require a minimum of 192×7 = 1344 memory banks!

Stride:
Handling Multidimensional Arrays in Vector Architectures

- Load/store units move groups of data between vector registers and memory
- The distance separating elements to be gathered into a single vector register is called the stride
- Three types of stride addressing
  - Unit stride
    - Contiguous (sequential) block of information in memory
    - Fastest : always possible to optimize this
  - Non unit (constant) stride
    - Harder to optimize memory system for all possible strides
    - Prime number of data banks makes it easier to support different strides at full bandwidth
  - Indexed (gather scatter)
    - Vector equivalent of register indirect
    - Good for sparse arrays of data
    - Increases number of programs that vectorize

Stride:

Handling Multidimensional Arrays in Vector Architectures

- The position in memory of adjacent elements in a vector may not be sequential.
- Consider this straightforward code for matrix multiply in C:

```
for (i = 0; i < 100; i = i + 1)
    for (j = 0; j < 100; j = j + 1) {
        A[i][j] = 0.0;
        for (k = 0; k < 100; k = k + 1)
        A[i][j] = A[i][j] + B[i][k] * D[k][j];
    }
```

Stride:
Handling Multidimensional Arrays in Vector Architectures

- We could vectorize the multiplication of each row of B with each column of D and strip-mine the inner loop with k as the index variable.
  - To do so, we must consider how to address adjacent elements in B and adjacent elements in D.
    - When an array is allocated memory, it is linearized and must be laid out in either row-major order (as in C) or column-major order (as in Fortran).
    - This linearization means that either the elements in the row or the elements in the column are not adjacent in memory.
  - For example, the preceding C code allocates in row-major order, so the elements of D that are accessed by iterations in the inner loop are separated by the row size times 8 (the number of bytes per entry) for a total of 800 bytes

Stride:

Handling Multidimensional Arrays in Vector Architectures

- This distance separating elements to be gathered into a single vector register is called the stride.
  - In the example, matrix D has a stride of 100 double words (800 bytes), and matrix B would have a stride of 1 double word (8 bytes).
- For column-major order, which is used by Fortran, the strides would be reversed.
  - Matrix D would have a stride of 1, while matrix B would have a stride of 100
- Thus, without reordering the loops, the compiler can't hide the long distances between successive elements for both B and D.

Stride:

Handling Multidimensional Arrays in Vector Architectures

- Once a vector is loaded into a vector register, it acts as if it had logically adjacent elements.
- Thus a vector processor can handle strides greater than one, called nonunit strides, using only vector load and vector store operations with stride capability.
  - This ability to access nonsequential memory locations and to reshape them into a dense structure is one of the major advantages of a vector architecture.

Stride:

Handling Multidimensional Arrays in Vector Architectures

- Supporting strides greater than one complicates the memory system.
- Once non-unit strides are introduced, it becomes possible to request accesses from the same bank frequently.
- When multiple accesses contend for a bank, a memory bank conflict occurs, thereby stalling one access.
- A bank conflict and thus a stall will occur if

$$\frac{\text{Number of banks}}{\text{Least common multiple (Stride, Number of banks)}} < \text{Bank busy time}$$

Example

- Suppose we have 8 memory banks with a bank busy time of 6 clocks and a total memory latency of 12 cycles.
- How long will it take to complete a 64-element vector load with a stride of 1? With a stride of 32?
- **Answer**
  - Because the number of banks is larger than the bank busy time, for a stride of 1, the load will take 12+64=76 clock cycles, or
    - 1.2 clock cycles per element.
  - The worst possible stride is a value that is a multiple of the number of memory banks, as in this case with a stride of 32 and 8 memory banks.
  - Every access to memory (after the first one) will collide with the previous access and will have to wait for the 6-clock-cycle bank busy time.
  - The total time will be 12+1+6 * 63=391 clock cycles, or
    - 6.1 clock cycles per element, slowing it down by a factor of 5!

Gather-Scatter:
Handling Sparse Matrices in Vector Architectures

- Important to have techniques to allow programs with sparse matrices to execute in vector mode.
  - In a sparse matrix, the elements of a vector are usually stored in some compacted form and then accessed indirectly.
- Assuming a simplified sparse structure, we might see code that looks like this:

```
for (i = 0; i < n;   i = i + 1)
    A[K[i]] = A[K[i]] + C[M[i]];
```

- This code implements a sparse vector sum on the arrays A and C, using index vectors K and M to designate the nonzero elements of A and C.

Gather-Scatter:
Handling Sparse Matrices in Vector Architectures

- The primary mechanism for supporting sparse matrices is gather-scatter operations using index vectors.
  - Goal is to support moving between a compressed representation and normal representation of a sparse matrix.
- A gather operation takes an index vector and fetches the vector whose elements are at the addresses given by adding a base address to the offsets given in the index vector.
  - The result is a dense vector in a vector register.
    - After these elements are operated on in a dense form, the sparse vector can be stored in an expanded form by a scatter store, using the same index vector.
- Hardware support for such operations is called gather-scatter, and it appears on nearly all modern vector processors.

Gather-Scatter:
Handling Sparse Matrices in Vector Architectures

- The RV64V instructions are vldi (load vector indexed or gather) and vsti (store vector indexed or scatter).
  - For example, if x5, x6, x7, and x28 contain the starting addresses of the vectors in the previous sequence, we can code the inner loop with vector instructions such as:

```
vsetdcfg    4*FP64          # 4 64b FP vector registers
vld         v0 , x7         # Load K[ ]
vldx        v1 , x5 , v0)   # Load A[K[ ]]
vld         v2 , x28        # Load M[ ]
vldi        v3 , x6 , v2)    # Load C[M[ ]]
vadd        v1 , v1 , v3    # Add them
vstx        v1 , x5 , v0)   # Store A[K[ ]]
vdisable                    # Disable vector registers
```

- This technique allows code with sparse matrices to run in vector mode.
- A simple vectorizing compiler could not automatically vectorize the preceding source code because the compiler would not know that the elements of K are distinct values, and thus that no dependences exist.

SIMD Instruction Set Extensions for Multimedia

- SIMD MMX started with observation that
  - many media applications operate on narrower data types than the 32-bit processors were optimized for.
- Graphics systems would use
  - 8 bits to represent each of the three primary colors plus 8 bits for transparency.
- Audio samples are usually represented with 8 or 16 bits.
- By partitioning the carry chains within, say, a 256-bit adder, a processor could perform simultaneous operations on short vectors of 32 8-bit operands, 16 16-bit operands, 8 32-bit operands, or 4 64-bit operands.

SIMD Instruction Set Extensions for Multimedia

- Typical multimedia SIMD instructions

Instruction category	Operands
Unsigned add/subtract	Thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit
Maximum/minimum	Thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit
Average	Thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit
Shift right/left	Thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit
Floating point	Sixteen 16-bit, eight 32-bit, four 64-bit, or two 128-bit

- In contrast to vector architectures, SIMD extensions have three major omissions, which make it harder for the compiler to generate SIMD code and increase the difficulty of programming in SIMD assembly language.

SIMD Instruction Set Extensions for Multimedia

- MM SIMD extensions fix the number of data operands in the opcode,
  - which has led to the addition of hundreds of instructions in the MMX, SSE, and AVX extensions of the x86 architecture.
  - Vector architectures have a vector-length register that specifies the number of operands for the current operation.
- MM SIMD did not offer the more sophisticated addressing modes of vector architectures (strided accesses and gather-scatter accesses).
  - These features increase the number of programs that a vector compiler can successfully vectorize
- MM SIMD usually did not offer the mask registers to support conditional execution of elements as in vector architectures

SIMD Instruction Set Extensions for Multimedia

- For the x86 architecture,
  - MMX instructions added in 1996 repurposed the 64-bit floating-point registers,
    - so the basic instructions could perform 8 8-bit operations or 4 16-bit operations simultaneously.
  - Streaming SIMD Extensions (SSE) successor in 1999 added 16 separate registers (XMM registers) that were 128 bits wide,
    - so now instructions could simultaneously perform 16 8-bit operations, 8 16-bit operations, or 4 32-bit operations.
  - Advanced Vector Extensions (AVX), added in 2010, doubled the width of the registers to 256 bits (YMM registers) and thereby offered
    - instructions that double the number of operations on all narrower data types

SIMD Instruction Set Extensions for Multimedia

- AVX instructions for x86 architecture useful in double-precision floating-point programs.

AVX instruction	Description
VADDPD	Add four packed double-precision operands
VSUBPD	Subtract four packed double-precision operands
VMULPD	Multiply four packed double-precision operands
VDIVPD	Divide four packed double-precision operands
VFMADDPD	Multiply and add four packed double-precision operands
VFMSUBPD	Multiply and subtract four packed double-precision operands
VCMPxx	Compare four packed double-precision operands for EQ, NEQ, LT, LE, GT, GE, ...
VMOVAPD	Move aligned four packed double-precision operands
VBROADCASTSD	Broadcast one double-precision operand to four locations in a 256-bit register

- Packed-double for 256-bit AVX means four 64-bit operands executed in SIMD mode.
- AVX includes instructions that shuffle 32-bit, 64-bit, or 128-bit operands within a 256-bit register.
  - For example, BROADCAST replicates a 64-bit operand four times in an AVX register.
- AVX also includes a large variety of fused multiply-add/subtract instructions

SIMD Instruction Set Extensions for Multimedia

- Why are MM SIMD extensions so popular?
  - they initially cost little to add to the standard arithmetic unit and they were easy to implement
  - they require scant extra processor state compared to vector architectures
  - a lot of memory bandwidth is needed to support a vector architecture, which many computers don't have
  - SIMD does not have to deal with problems in virtual memory when a single instruction can generate 32 memory accesses and any of which can cause a page fault
    - original SIMD extensions used separate data transfers per SIMD group of operands that are aligned in memory, and so they cannot cross page boundaries

Example SIMD Code

- This example shows RISC-V SIMD code for the DAXPY loop, with the changes to the RISC-V code for SIMD underlined.
  - Starting addresses of X and Y are in x5 and x6, respectively.

```
        fld       f0 , a          # Load scalar a
        splat .4D     f0 , f0         # Make 4 copies of a
        addi      x28 , x5 , #256     # Last address to load
Loop:   fld.4D    f1 , 0(x5)      # Load X[i] ... X[i+3]
        fmul.4D       f1 , f1 , f0    # a x X[i] ... a x X[i+3]
        fld.4D    f2 , 0(x6)      # Load Y[i] ... Y[i+3]
        fadd.4D   f2 , f2 , f1    # a x X[i]+Y[i]...
                                  # a x X[i+3]+Y[i+3]
        fsd.4D    f2 , 0(x6)      # Store Y[i]... Y[i+3]
        addi      x5 , x5 , #32   # Increment index to X
        addi      x6 , x6 , #32   # Increment index to Y
        bne       x28 , x5 , Loop     # Check if done
```

Programming Multimedia SIMD Architectures

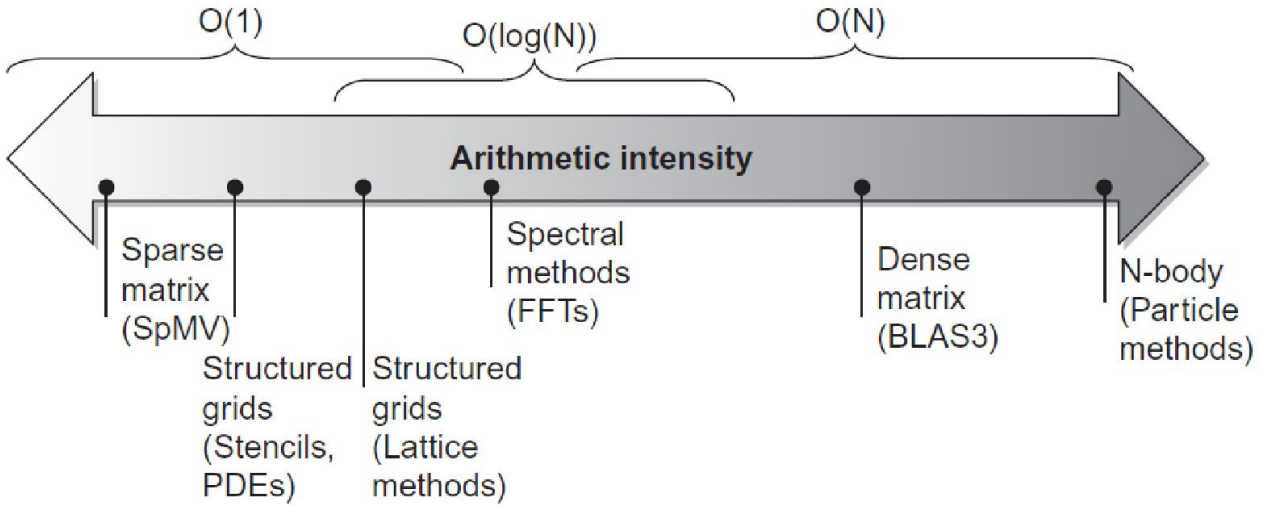
- Easiest way to use SIMD MMX instructions has been through libraries or by writing in assembly language.
- Recent extensions have become more regular, giving compilers a more reasonable target.
  - By borrowing techniques from vectorizing compilers, compilers are starting to produce SIMD instructions automatically.
  - For example, advanced compilers today can generate SIMD fp instructions to deliver much higher performance for scientific codes.
    - However, programmers must be sure to align all the data in memory to the width of the SIMD unit on which the code is run to prevent the compiler from generating scalar instructions for otherwise vectorizable code.

Roofline Visual Performance Model

- Roofline model
  - Visual, intuitive way to compare potential floating-point performance of variations of SIMD architectures
  - horizontal and diagonal lines of the graphs it produces give this simple model its name and indicate its value
  - It ties together floating-point performance, memory performance, and arithmetic intensity in a two-dimensional graph.
- Arithmetic intensity
  - The ratio of fp operations per byte of memory accessed.
    - can be calculated by taking the total number of fp operations for a program divided by the total number of data bytes transferred to main memory during program execution.

Roofline Visual Performance Model

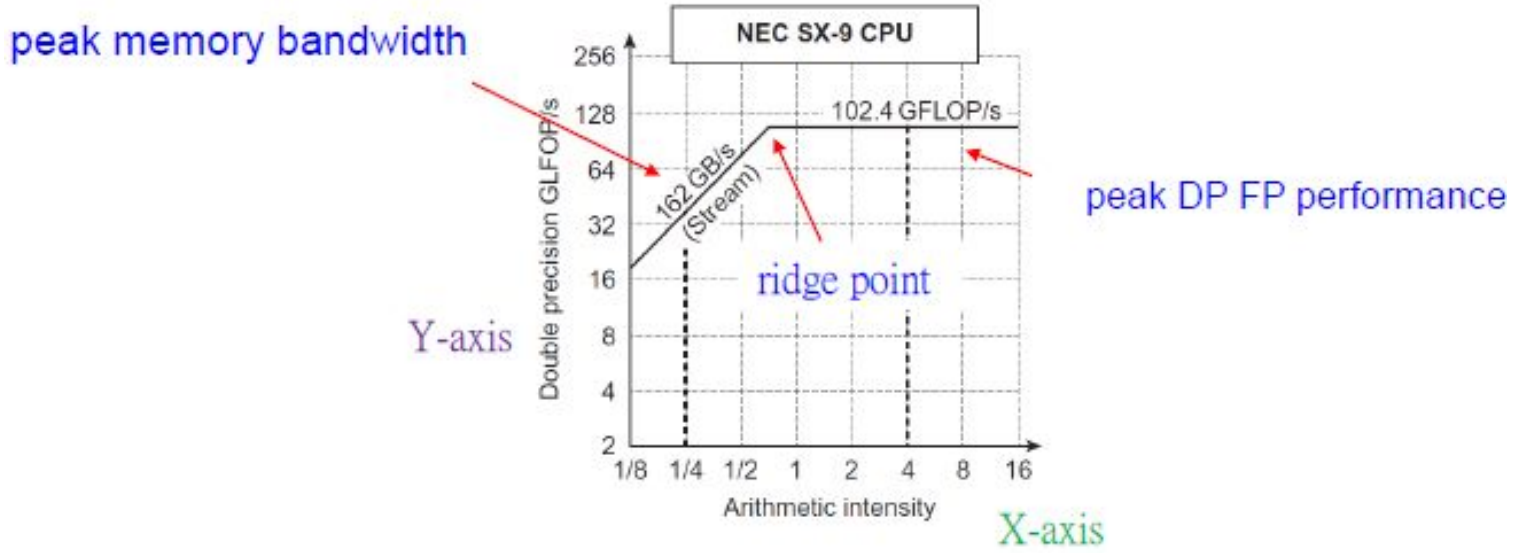
- Following figure shows the relative arithmetic intensity of several example kernels.



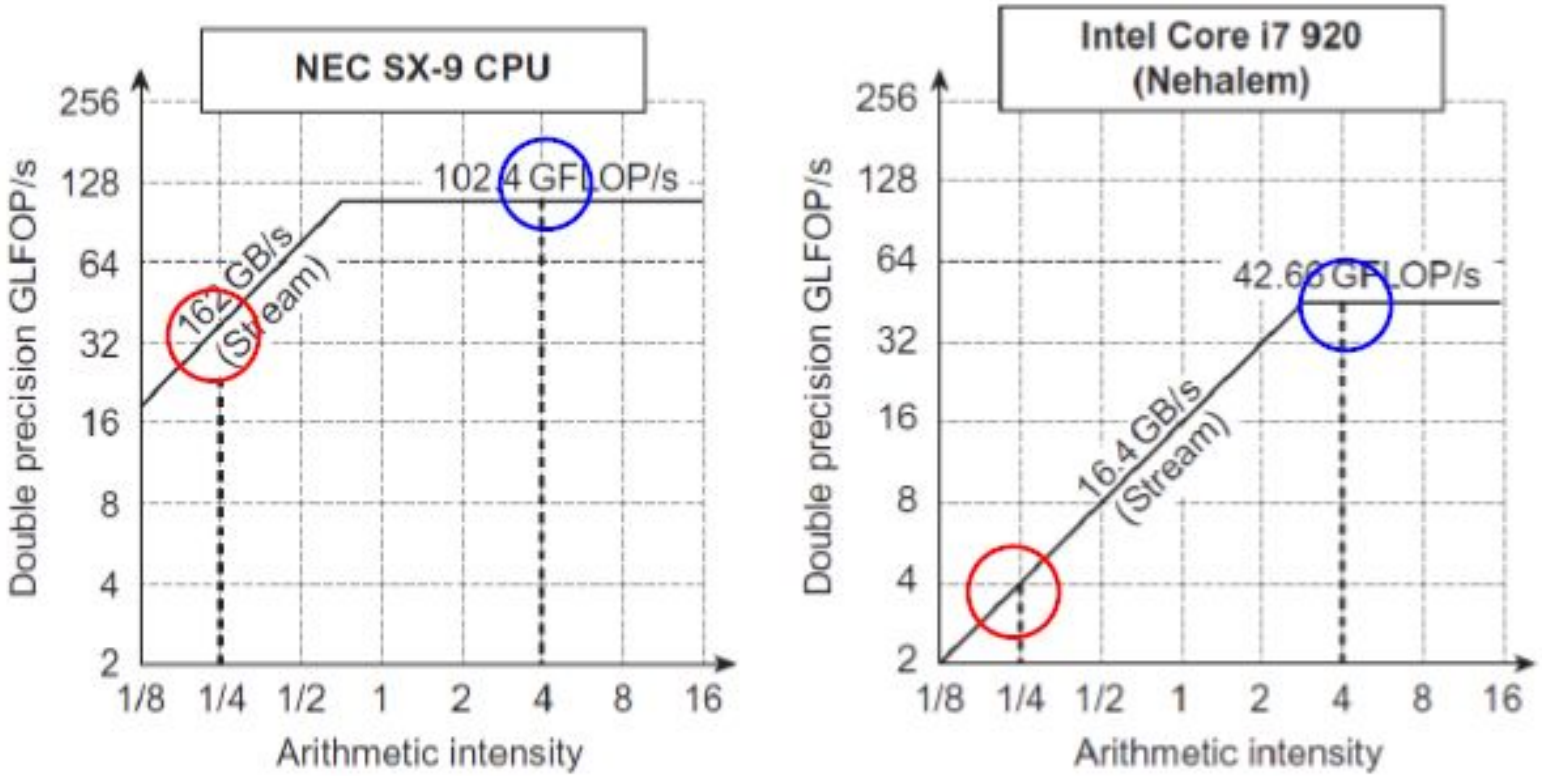
- Arithmetic intensity, specified as the number of fp operations to run the program divided by the number of bytes accessed in main memory
  - Some kernels have an arithmetic intensity that scales with problem size, such as a dense matrix, but there are many kernels with arithmetic intensities independent of problem size.

Roofline Model Examples

- The "Roofline" sets an upper bound on performance of a kernel depending on its arithmetic intensity.
  - Y axis: attainable fp performance (GFLOPs/sec)
    Attainable GFLOPs/sec = (Peak Memory BW × Arithmetic Intensity, Peak Floating Point Perf.)
  - X axis: arithmetic intensity (1/8 to 16 FLOP/DRAM byte accessed)



Comparisons on Roofline Models



- The dashed vertical lines at an arithmetic intensity of 4 FLOP/byte:
  - The SX-9 at 102.4 FLOP/s is 2.4× faster than the Core i7 at 42.66 GFLOP/s.
- At an arithmetic intensity of 1/4 FLOP/byte:
  - The SX-9 at 40.5 GFLOP/s is 10× faster than the Core i7 at 4.1 GFLOP/s.

Roofline Visual Performance Model

- How could we plot the peak memory performance?
- Because the X-axis is FLOP/byte and the Y-axis is FLOP/s, bytes/s is just a diagonal line at a 45-degree angle in the figure.
  - Thus we can plot a third line that gives the maximum fp performance that the memory system of that computer can support for a given arithmetic intensity.
- We can express the limits as a formula to plot these lines in the graphs in previous slide

  Attainable GFLOPs/s = Min (Peak Memory BW × Arithmetic Intensity, Peak fp Perf.)
- Roofline sets an upper bound on performance of a kernel depending on its arithmetic intensity.

Graphics Processing Units

- A highly parallel, highly multithreaded multiprocessor optimized for visual computing.
  - GPU generates 2D and 3D graphics, images, and video that enable window based operating systems, graphical user interfaces, video games, visual imaging applications, and video
- To provide real-time visual interaction with computed objects via graphics, images, and video, the GPU has a unified graphics and computing architecture that serves as both a programmable graphics processor and a scalable parallel computing platform.
- PCs and game consoles combine a GPU with a CPU to form heterogeneous systems.

Graphics Processing Units

- Graphics Processing Unit (GPU)
  - A processor optimized for 2D and 3D graphics, video, visual computing, and display.
- Visual computing
  - A mix of graphics processing and computing that lets you visually interact with computed objects via graphics, images, and video.
- Heterogeneous system
  - A system combining different processor types.
    - A PC is a heterogeneous CPU–GPU system.

A Brief History of GPU Evolution

- Graphics on a PC were performed by a Video Graphics Array (VGA) controller (20 years ago)
  - a memory controller and display generator connected to some DRAM
- 1990s, more functions could be added to the VGA controller
- By 1997, incorporate some three-dimensional (3D) acceleration functions
- In 2000, single chip graphics processor incorporated almost every detail of the traditional high-end workstation graphics pipeline
  - The term GPU was coined to denote that the graphics device had become a processor

GPU Graphics Trends

- GPUs and their associated drivers implement the OpenGL and DirectX models of graphics processing.
  - OpenGL is an open standard for 3D graphics programming available for most computers.
  - DirectX is a series of Microsoft multimedia programming interfaces.
- Since these APIs have well-defined behavior, it is possible to build effective hardware acceleration of the graphics processing functions defined by the APIs.
  - API (Application Programming Interface)
    - A set of function and data structure definitions providing an interface to a library of functions.

GPU Evolves into Scalable Parallel Processor

- GPUs have evolved functionally from hardwired, limited capability VGA controllers to programmable parallel processors
- This evolution has proceeded by changing the logical (API-based) graphics pipeline to incorporate programmable elements and also by making the underlying hardware pipeline stages less specialized and more programmable.
- Disparate programmable pipeline elements merged into one unified array of many programmable processors

CUDA and GPU Computing

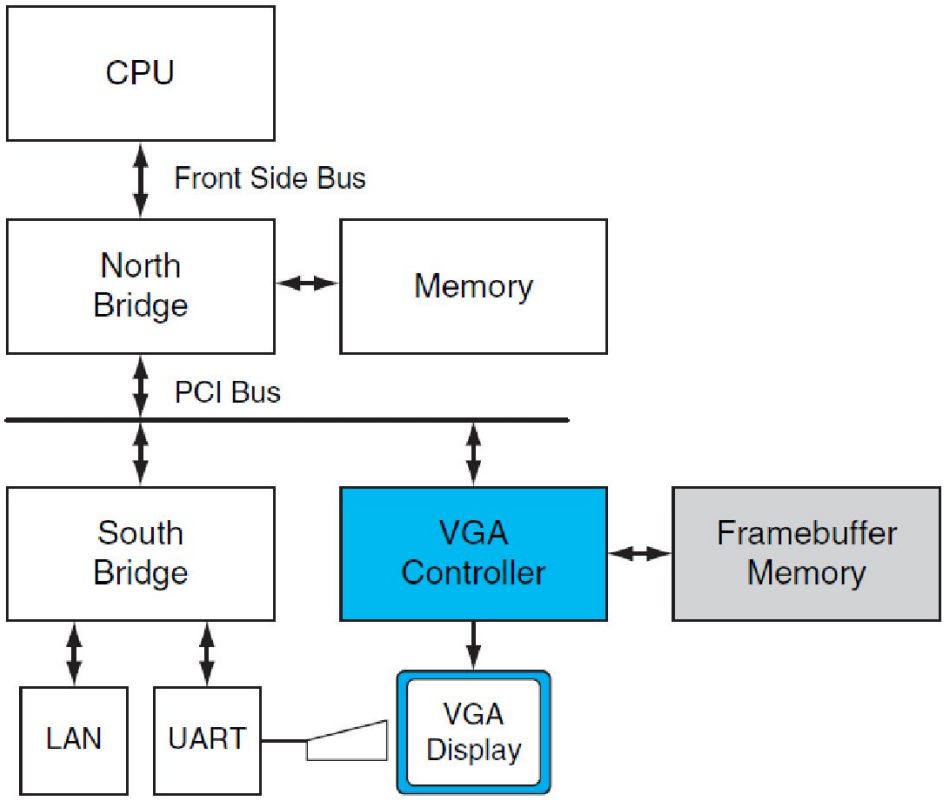
- GPU computing
  - Using a GPU for computing via a parallel programming language and API.
- GPGPU (General Purpose Computation on GPU)
  - Using a GPU for general-purpose computation via a traditional graphics API and graphics pipeline.
- CUDA (Compute Unified Device Architecture)
  - A scalable parallel programming model and language based on C/C++.
  - It is a parallel programming platform for GPUs and multicore CPUs.

Compute Unified Device Architecture

- CUDA programming model has an SPMD (Single-Program Multiple Data) software style, in which a programmer writes a program for one thread that is instanced and executed by many threads in parallel on the multiple processors of the GPU.
- CUDA also provides a facility for programming multiple CPU cores as well,
  - so CUDA is an environment for writing parallel programs for the entire heterogeneous computer system.

GPU System Architectures

- The Historical PC (circa 1990)

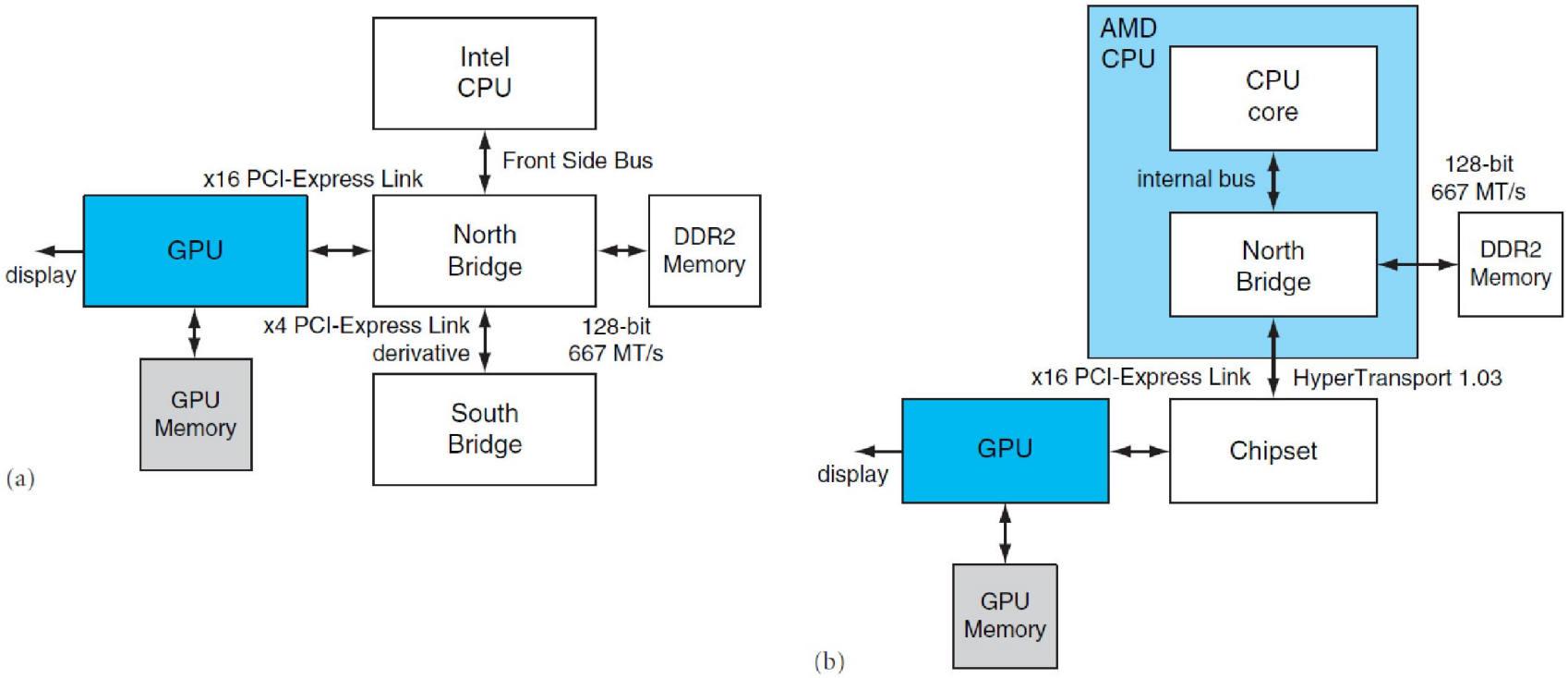


- North bridge contains high-bandwidth interfaces, connecting the CPU, memory, and PCI bus.
- South bridge contains legacy interfaces and devices:
- ISA bus (audio, LAN), interrupt controller; DMA controller; time/counter.

- The display was driven by a simple frame buffer subsystem known as a VGA which was attached to the PCI bus

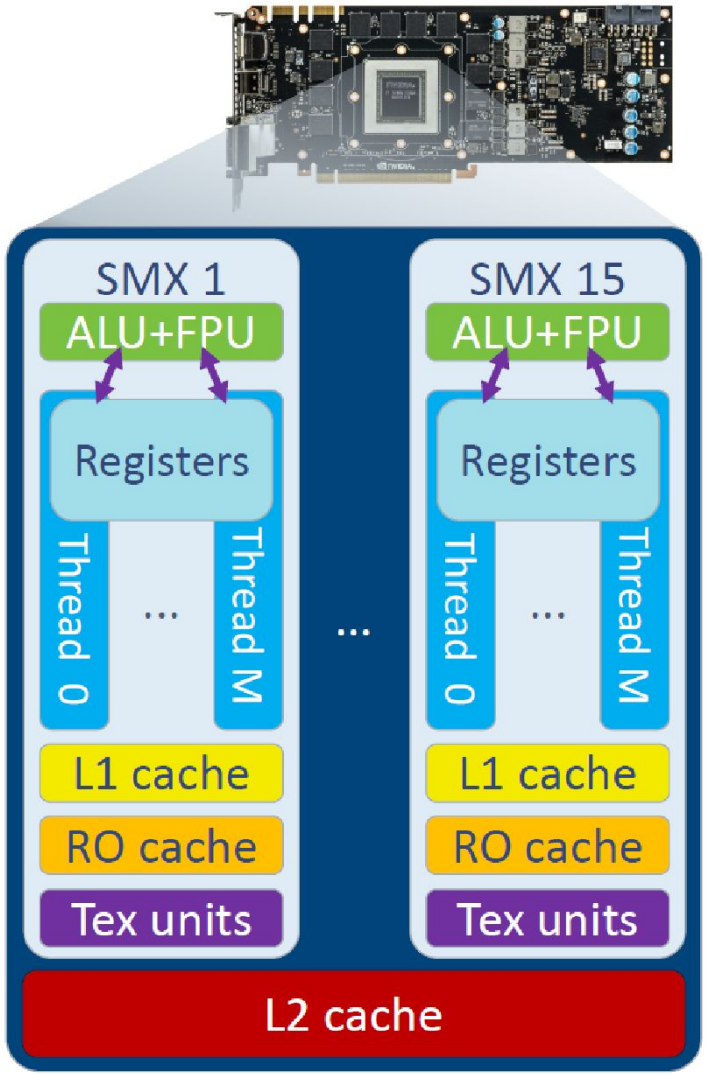
GPU System Architectures

- Contemporary PCs with Intel and AMD CPUs
  - Characterized by a separate GPU (discrete GPU) and CPU with respective memory subsystems.



Many-core GPU architecture

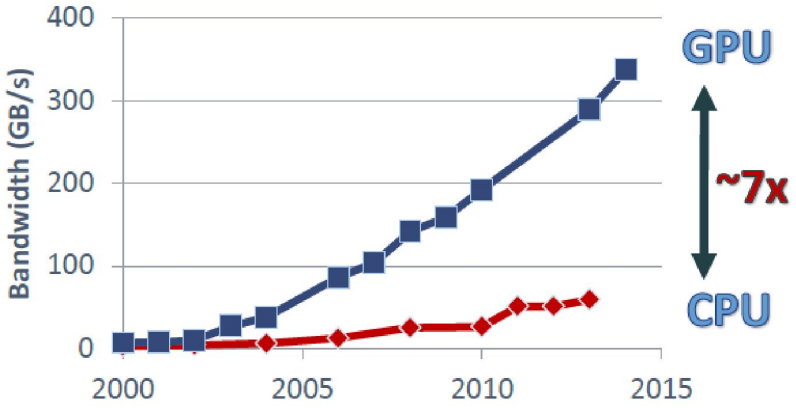
- A single core (streaming multiprocessor, SMX)
  - L1 cache, Read only cache, texture units
  - 6 32-wide SIMD units (192 total, single precision)
  - Up-to 64 warps simultaneously (hardware warps)
    - Like hyper-threading, but a warp is 32-wide SIMD
- Optimal number of FLOPS per clock cycle:
  - 32x: 32-way SIMD
  - 2x: Fused multiply add
  - 6x: 6 SIMD units per core
  - 15x: 15 cores
  - Sum: 5760!

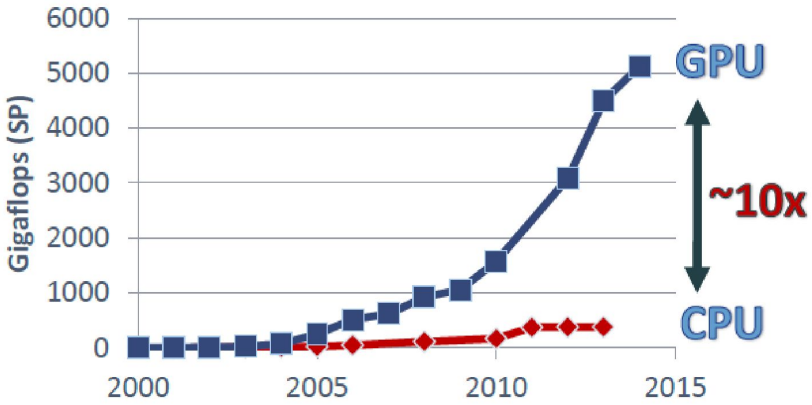


Simplified schematic of GPU design

Massive Parallelism

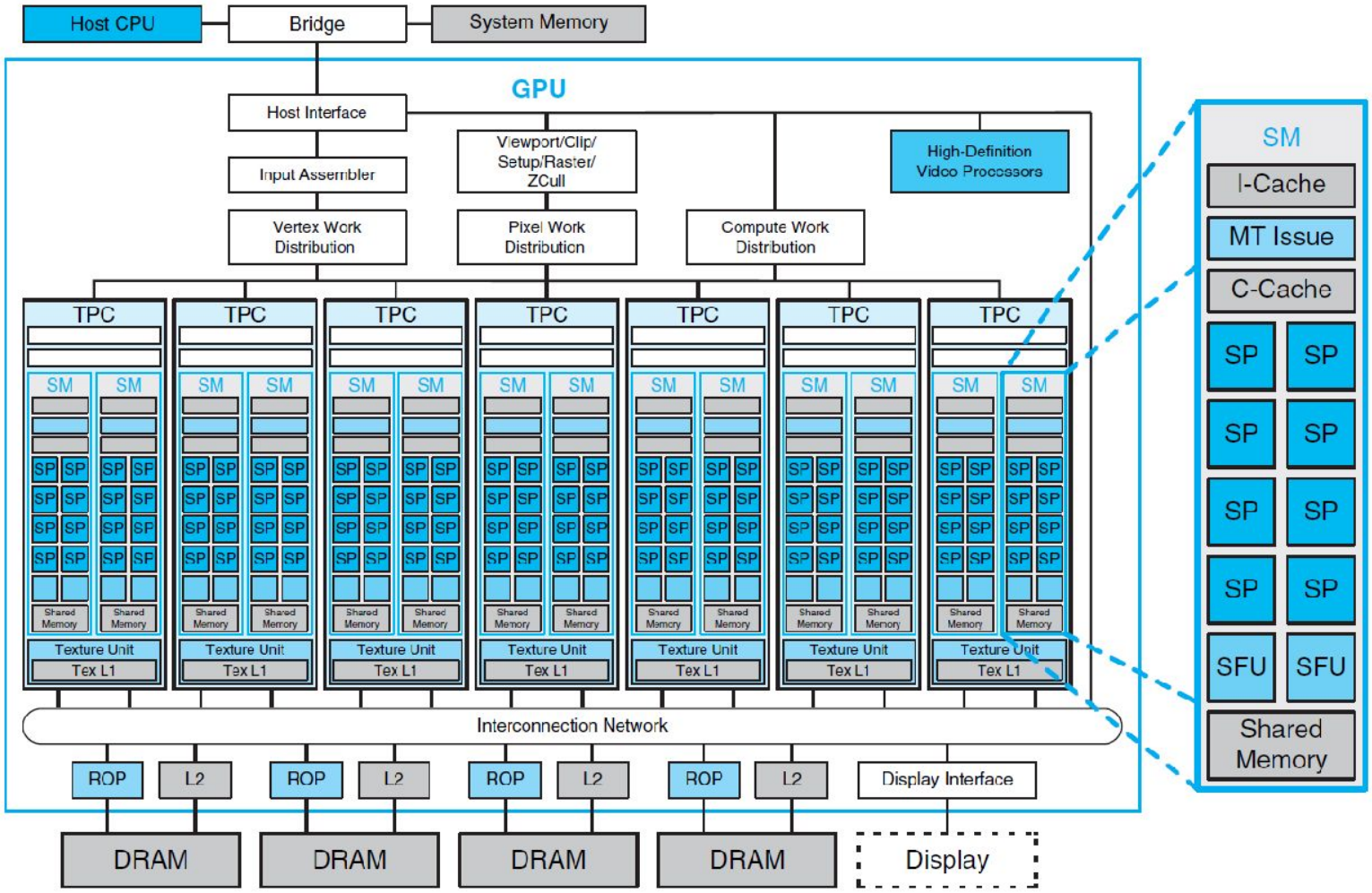
- Up-to 5760 floating point operations in parallel!
- 5-10 times as power efficient as CPUs!





GPU System Architectures

- Basic unified GPU architecture



GPU System Architectures

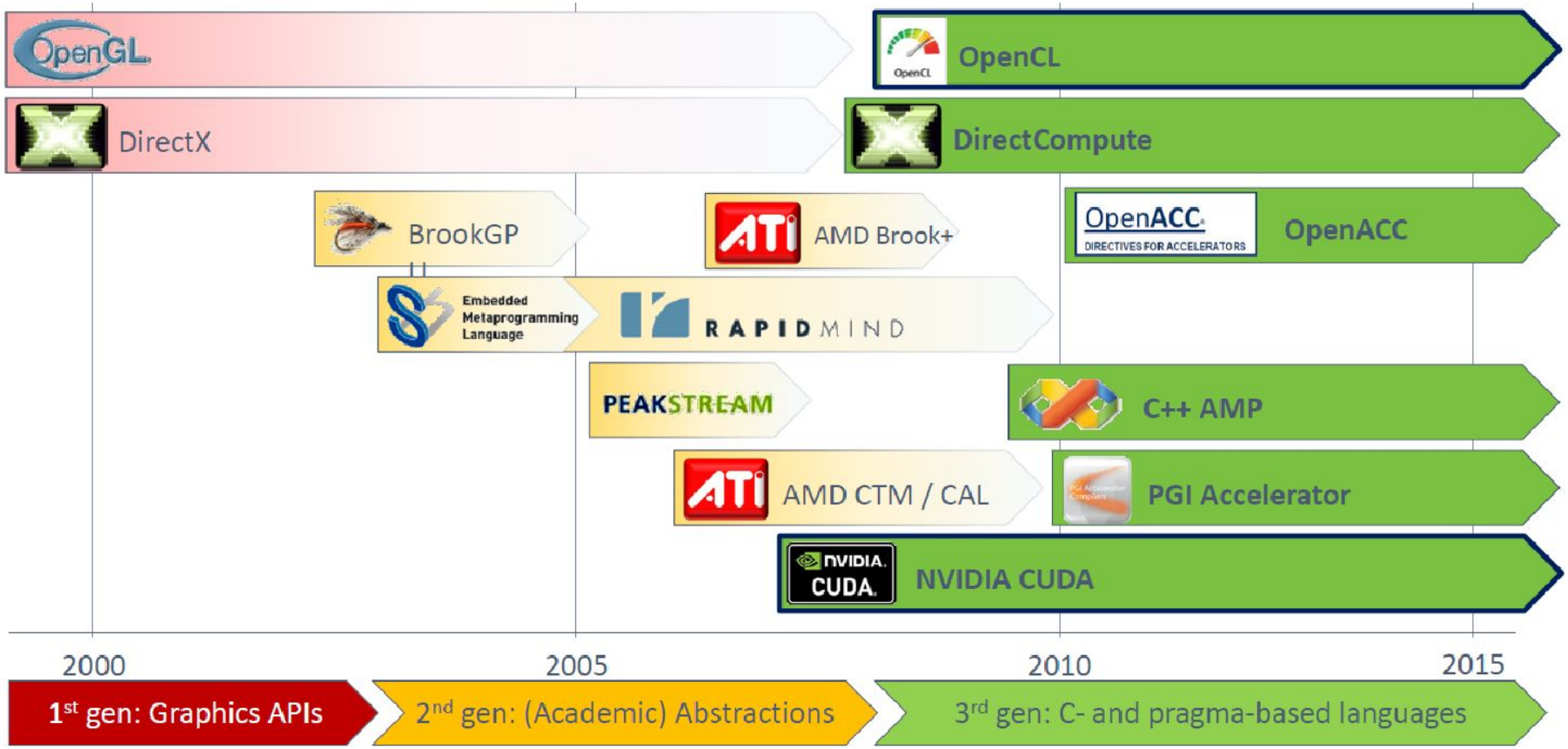
- 112 streaming processor (SP) cores
  - organized in 14 streaming multiprocessors (SMs);
  - the cores are highly multithreaded.
- It has the basic Tesla architecture of an NVIDIA GeForce 8800.
  - The processors connect with 4 64-bit-wide DRAM partitions via an interconnection network.
  - Each SM has 8 SP cores, 2 special function units (SFUs), instruction and constant caches, a multithreaded instruction unit, and a shared memory.

Programming the GPU

- Challenges for the GPU programmer:
  - getting good performance on the GPU
  - coordinating the scheduling of computation on the system processor and the GPU
  - transfer of data between system memory and GPU memory
- GPUs have virtually every type of parallelism that can be captured by the programming environment:
  - multithreading, MIMD, SIMD, and even instruction-level
- NVIDIA develop a C-like language and programming environment that would improve the productivity of GPU programmers:
  - CUDA (Compute Unified Device Architecture)
    - CUDA produces C/C++ for the system processor (host) and a C and C++ dialect for the GPU
- A similar programming language is OpenCL, which several companies are developing to offer a vendor-independent language for multiple platforms

Programming the GPU

- GPU Programming Languages

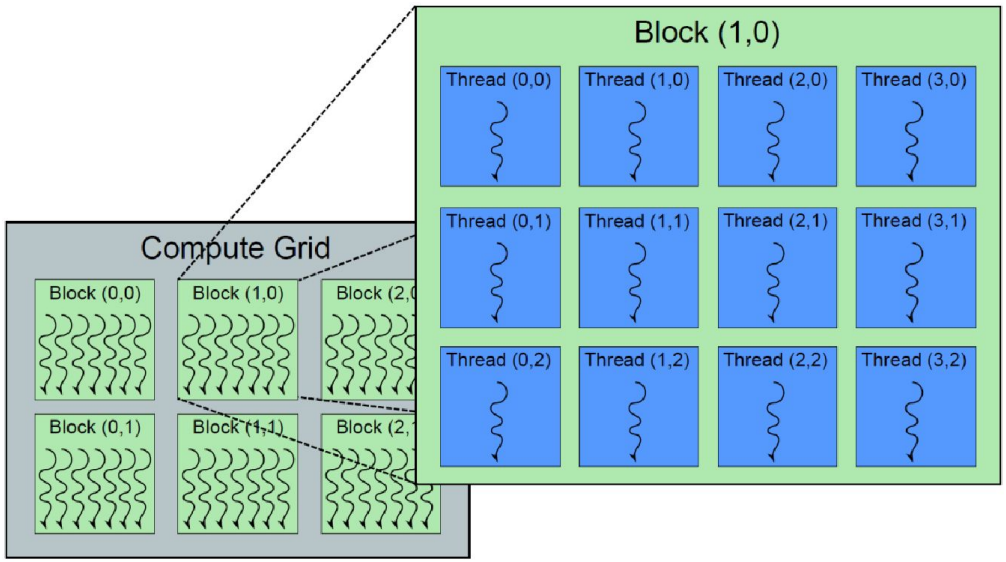


Threads and Blocks

- NVIDIA decided that the unifying theme of all these forms of parallelism is the CUDA Thread
  - A thread is associated with each data element
  - Threads are organized into blocks (Thread Block)
  - Blocks are organized into a grid
- GPU hardware handles thread management, not applications or OS
- Hardware that executes a whole block of threads is called multithreaded SIMD Processor
- NVIDIA classifies the CUDA programming model as single instruction, multiple thread (SIMT)

Grids and blocks in CUDA

- Two-layered parallelism
  - A block consists of threads:



- Threads within the same block can cooperate and communicate
  - A grid consists of blocks:
- All blocks run independently.
  - Blocks and grid can be 1D, 2D, and 3D
- Global synchronization and communication is only possible between kernel launches
  - Expensive, and should be avoided if possible

Programming the GPU

- Computing $y = ax + y$ with a serial loop (conventional C code for the DAXPY loop):

```
// Invoke DAXPY
daxpy(n, 2.0, x, y);
// DAXPY in C
void daxpy(int n, double a, double *x, double *y)
{
   for (int i = 0; i < n; ++i)
   y[i] = a*x[i] + y[i];
}
```

- has a loop where each iteration is independent from the others,
  - allowing the loop to be transformed straightforwardly into a parallel code where each loop iteration becomes a separate thread.

Programming the GPU

- Computing y = ax + y in parallel using CUDA:

```
// Invoke DAXPY with 256 threads per Thread Block
__host__
int nblocks = (n+ 255) / 256;
        daxpy<<<nblocks, 256>>>(n, 2.0, x, y);
// DAXPY in CUDA
__global__
void daxpy(int n, double a, double *x, double *y)
{
        int i = blockIdx.x*blockDim.x + threadIdx.x;
        if (i < n) y[i] = a*x[i] + y[i];
}
```

- n threads, one per vector element, with 256 CUDA Threads per Thread Block in a multithreaded SIMD Processor.
- GPU function starts by calculating the corresponding element index i based on the block ID, the number of threads per block, and the thread ID.
  - As long as this index is within the array (i < n), it performs the multiply and add.

Example: Adding two matrices in CUDA

- We want to add two matrices, a and b, and store the result in c.

$$\begin{bmatrix} 1 & 3 \\ 1 & 0 \\ 1 & 2 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 7 & 5 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 1+0 & 3+0 \\ 1+7 & 0+5 \\ 1+2 & 2+1 \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ 8 & 5 \\ 3 & 3 \end{bmatrix}$$

- For best performance, loop through one row at a time (sequential memory access pattern)

```
void   addFunctionCPU(float* c, float* a, float* b,
                      unsigned int cols, unsigned int rows) {
    for    (unsigned int j=0; j<rows; ++j) {
      for    (unsigned int i=0; i<cols; ++i) {
               unsigned int k = j*cols + i;
               c[k] = a[k] + b[k];
      }
    }
}
```

Example: Adding two matrices in CUDA

```
__global__ void addMatricesKernel(float* c, float* a, float* b,
                unsigned int cols, unsigned int rows) {
    //Indexing calculations
    unsigned int global_x = blockIdx.x*blockDim.x + threadIdx.x;
    unsigned int global_y = blockIdx.y*blockDim.y + threadIdx.y;
    unsigned int k = global_y*cols + global_x;

    //Actual addition
    c[k] = a[k] + b[k];
}

void addFunctionGPU(float* c, float* a, float* b,
unsigned int cols, unsigned int rows) {
    dim3   block(8, 8);
    dim3   grid(cols/8, rows/8);
    ... //More code here: Allocate data on GPU, copy CPU data to GPU
    addMatricesKernel<<<grid, block>>>(gpu_c, gpu_a, gpu_b, cols, rows);
    ... //More code here: Download result from GPU to CPU
}
```

GPU function

Indices

Implicit double for loop
for (int blockIdx.x = 0;
blockIdx.x < grid.x;
blockIdx.x) { …

Run on GPU

NVIDIA GPU Architecture

- Similarities to vector machines:
  - Works well with data-level parallel problems
  - Scatter-gather transfers
  - Mask registers
  - Large register files

- Differences:
  - No scalar processor
  - Uses multithreading to hide memory latency
  - Has many functional units, as opposed to a few deeply pipelined units like a vector processor

Example

- Code that works over all elements is the grid
- Thread blocks break this down into manageable sizes
  - 512 threads per block
- SIMD instruction executes 32 elements at a time
- Thus grid size = 16 blocks
- Block is analogous to a strip-mined vector loop with vector length of 32
- Block is assigned to a multithreaded SIMD processor by the thread block scheduler
- Current-generation GPUs have 7-15 multithreaded SIMD processors

Quick guide to GPU terms

Type	Descriptive name	Closest old term outside of GPUs	Official CUDA/NVIDIA GPU term	Short explanation
Program abstractions	Vectorizable Loop	Vectorizable Loop	Grid	A vectorizable loop, executed on the GPU, made up of one or more Thread Blocks (bodies of vectorized loop) that can execute in parallel
	Body of Vectorized Loop	Body of a (Strip-Mined) Vectorized Loop	Thread Block	A vectorized loop executed on a multithreaded SIMD Processor, made up of one or more threads of SIMD instructions. They can communicate via local memory
	Sequence of SIMD Lane Operations	One iteration of a Scalar Loop	CUDA Thread	A vertical cut of a thread of SIMD instructions corresponding to one element executed by one SIMD Lane. Result is stored depending on mask and predicate register
Machine object	A Thread of SIMD Instructions	Thread of Vector Instructions	Warp	A traditional thread, but it only contains SIMD instructions that are executed on a multithreaded SIMD Processor. Results stored depending on a per-element mask
	SIMD Instruction	Vector Instruction	PTX Instruction	A single SIMD instruction executed across SIMD Lanes

Quick guide to GPU terms

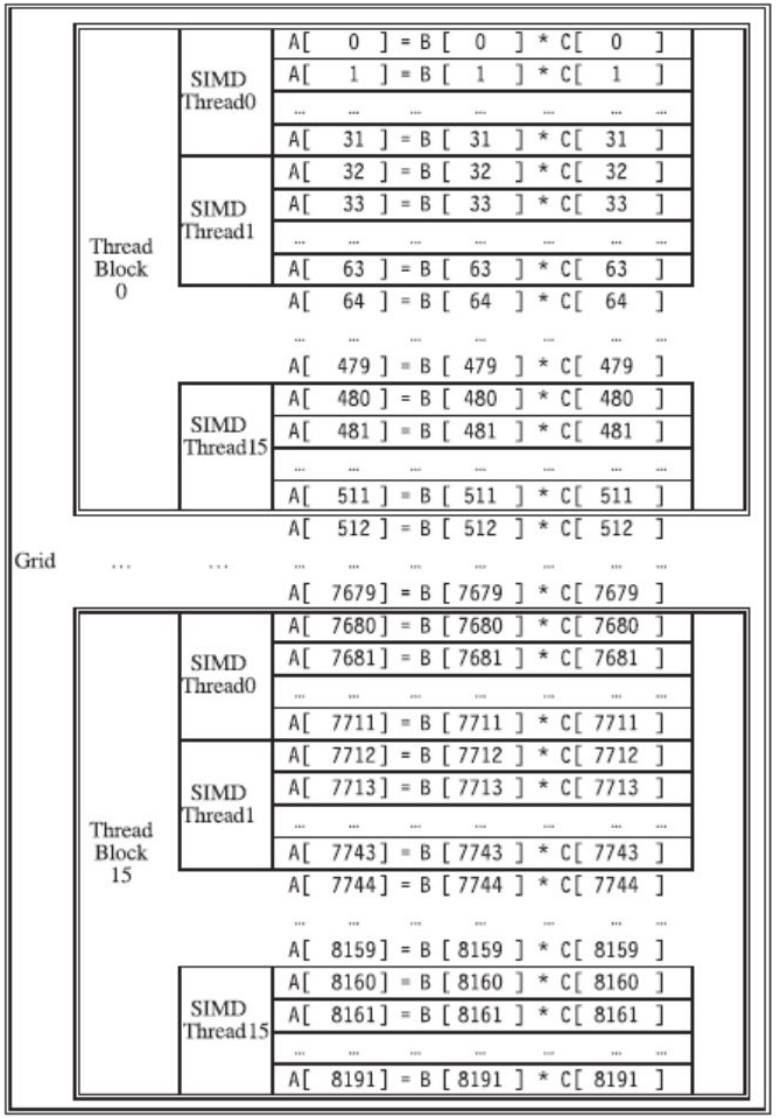
Processing hardware	Multithreaded SIMD Processor	(Multithreaded) Vector Processor	Streaming Multiprocessor	A multithreaded SIMD Processor executes threads of SIMD instructions, independent of other SIMD Processors
	Thread Block Scheduler	Scalar Processor	Giga Thread Engine	Assigns multiple Thread Blocks (bodies of vectorized loop) to multithreaded SIMD Processors
	SIMD Thread Scheduler	Thread Scheduler in a Multithreaded CPU	Warp Scheduler	Hardware unit that schedules and issues threads of SIMD instructions when they are ready to execute; includes a scoreboard to track SIMD Thread execution
	SIMD Lane	Vector Lane	Thread Processor	A SIMD Lane executes the operations in a thread of SIMD instructions on a single element. Results stored depending on mask
Memory hardware	GPU Memory	Main Memory	Global Memory	DRAM memory accessible by all multithreaded SIMD Processors in a GPU
	Private Memory	Stack or Thread Local Storage (OS)	Local Memory	Portion of DRAM memory private to each SIMD Lane
	Local Memory	Local Memory	Shared Memory	Fast local SRAM for one multithreaded SIMD Processor, unavailable to other SIMD Processors
	SIMD Lane Registers	Vector Lane Registers	Thread Processor Registers	Registers in a single SIMD Lane allocated across a full Thread Block (body of vectorized loop)

Terminology

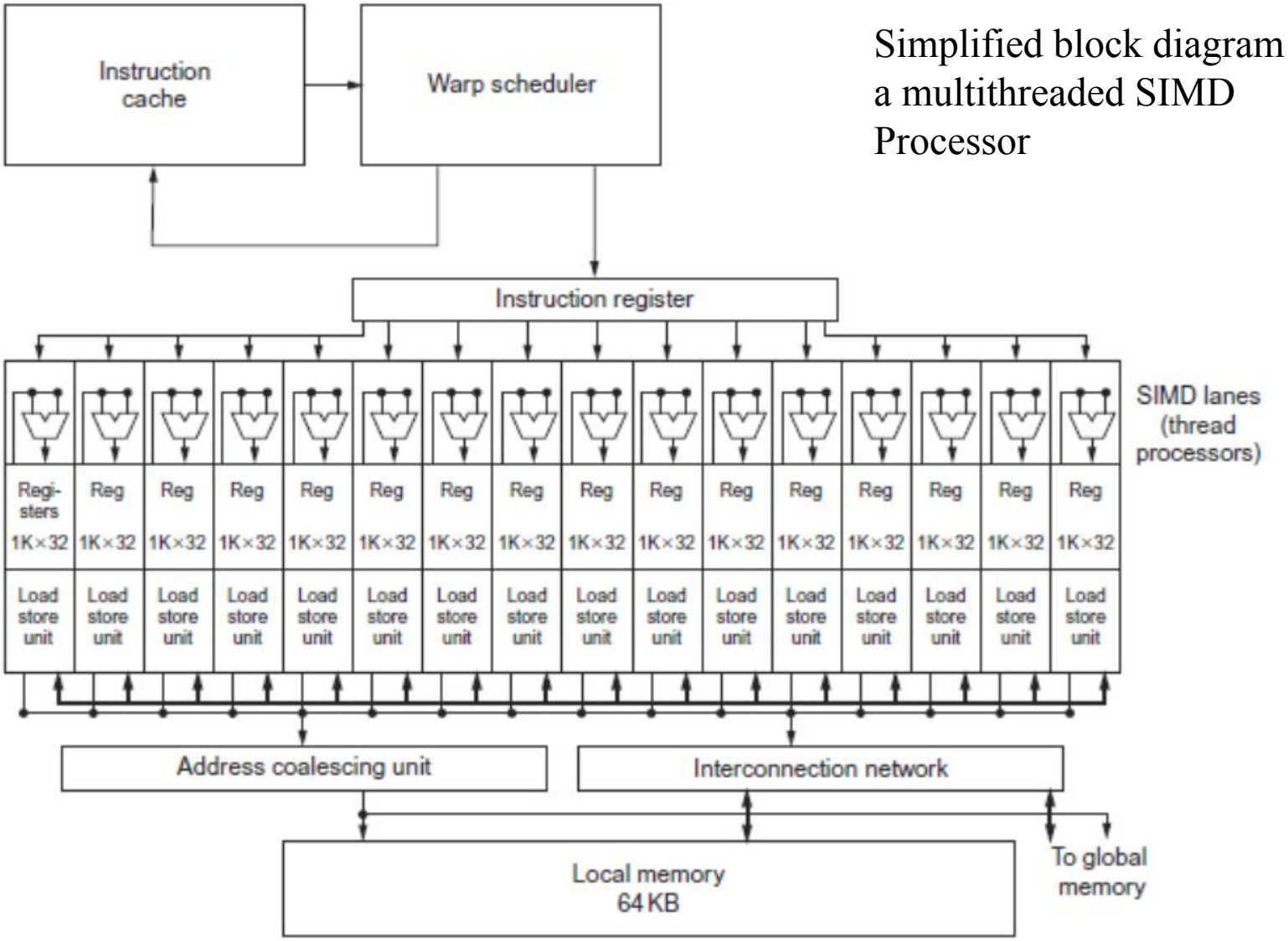
- A Grid is the code that runs on a GPU that consists of a set of Thread Blocks.
- Each thread is limited to 64 registers
- Groups of 32 threads combined into a SIMD thread or “warp”
  - Mapped to 16 physical lanes
- Up to 32 warps are scheduled on a single SIMD processor
  - Each warp has its own PC
  - Thread scheduler uses scoreboard to dispatch warps
  - By definition, no data dependencies between warps
  - Dispatch warps into pipeline, hide memory latency
- Thread block scheduler schedules blocks to SIMD processors
- Within each SIMD processor:
  - 32 SIMD lanes
  - Wide and shallow compared to vector processors

Example

- Mapping of a Grid (vectorizable loop), Thread Blocks (SIMD basic blocks), and threads of SIMD instructions to a vector-vector multiply, with each vector being 8192 elements long.
- Each thread of SIMD instructions calculates 32 elements per instruction,
- Each Thread Block contains 16 threads of SIMD instructions and the Grid contains 16 Thread Blocks.
- The hardware Thread Block Scheduler assigns Thread Blocks to multithreaded SIMD Processors, and the hardware Thread Scheduler picks which thread of SIMD instructions to run each clock cycle within a SIMD Processor.
- Only SIMD Threads in the same Thread Block can communicate via local memory.
  - The maximum number of SIMD Threads that can execute simultaneously per Thread Block is 32 for Pascal GPUs.



GPU Organization



Simplified block diagram of a multithreaded SIMD Processor

Pascal P100 GPU

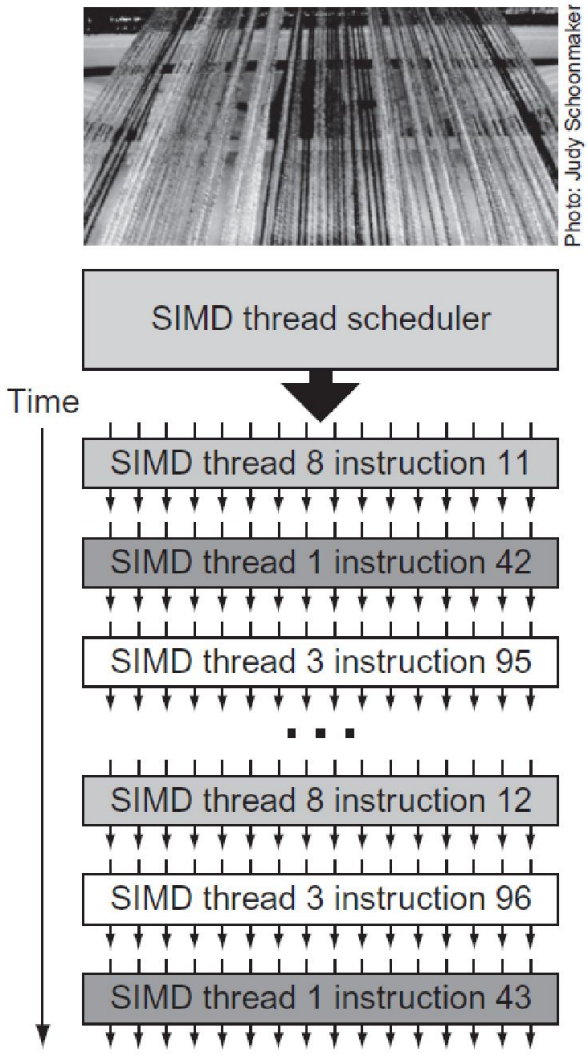
- Full-chip block diagram of the Pascal P100 GPU

Pascal P100 GPU

- It has 56 multithreaded SIMD Processors,
  - each with an L1 cache and local memory,
- 32 L2 units, and a memory-bus width of 4096 data wires.
  - It has 60 blocks, with four spares to improve yield.
- The P100 has 4 HBM2 ports supporting up to 16 GB of capacity.
- It contains 15.4 billion transistors.

Scheduling of threads of SIMD instructions

- The scheduler selects a ready thread of SIMD instructions and issues an instruction synchronously to all the SIMD Lanes executing the SIMD Thread.
- Because threads of SIMD instructions are independent, the scheduler may select a different SIMD Thread each time.



NVIDIA Instruction Set Arch.

- ISA is an abstraction of the hardware instruction set
  - Parallel Thread Execution (PTX)
    - provides a stable instruction set for compilers
    - as compatibility across generations of GPUs.
    - Format of a PTX instruction is
      - opcode.type d,a,b,c;
        - where d is the destination operand; a, b, and c are source operands;
        - operation type is one of the following:

Type	.type specifier
Untyped bits 8, 16, 32, and 64 bits	.b8, .b16, .b32, .b64
Unsigned integer 8, 16, 32, and 64 bits	.u8, .u16, .u32, .u64
Signed integer 8, 16, 32, and 64 bits	.s8, .s16, .s32, .s64
Floating Point 16, 32, and 64 bits	.f16, .f32, .f64

NVIDIA Instruction Set Arch.

- Uses virtual registers
- Translation to machine code is performed in software

- Example:
  - Following sequence of PTX instructions is for one iteration of DAXPY

```
shl.s32       R8, blockIdx, 9 ; Thread Block ID * Block size (512 or 29)
add.s32       R8, R8, threadIdx      ; R8 = i = my CUDA thread ID
ld.global.f64   RD0, [X+R8]        ; RD0 = X[i]
ld.global.f64   RD2, [Y+R8]        ; RD2 = Y[i]
mul.f64 R0D, RD0, RD4     ; Product in RD0 = RD0 * RD4 (scalar a)
add.f64 R0D, RD0, RD2     ; Sum in RD0 = RD0 + RD2 (Y[i])
st.global.f64 [Y+R8], RD0  ; Y[i] = sum (X[i]*a + Y[i])
```

- CUDA programming model assigns one CUDA Thread to each loop iteration and offers a unique identifier number to each Thread Block (blockIdx) and one to each CUDA Thread within a block (threadIdx).

Conditional Branching

- Like vector architectures, GPU branch hardware uses internal masks
- Also uses
  - Branch synchronization stack
    - Entries consist of masks for each SIMD lane
    - I.e. which threads commit their results (all threads execute)
  - Instruction markers to manage when a branch diverges into multiple execution paths
    - Push on divergent branch
  - …and when paths converge
    - Act as barriers
    - Pops stack
- Per-thread-lane 1-bit predicate register, specified by programmer

Example

- The code for a conditional statement

```
if (X[i] != 0)
   X[i] = X[i] – Y[i];
else X[i] = Z[i];
```

- This IF statement could compile to the following PTX instructions:

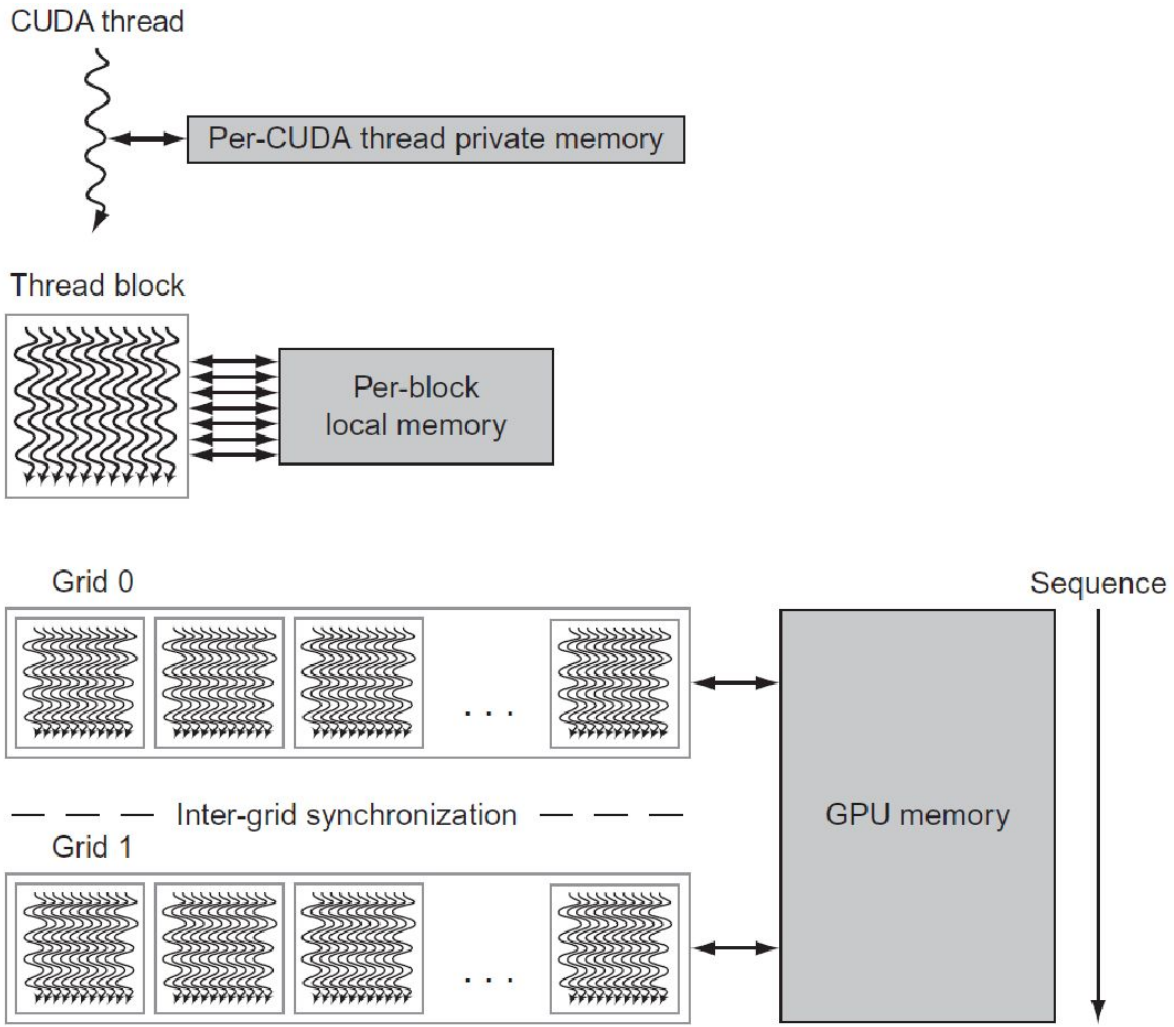
```
    ld.global.f64   RD0, [X+R8]  ; RD0 = X[i]
    setp.neq.s32    P1, RD0, #0    ; P1 is predicate register 1
    @!P1, bra       ELSE1, *Push       ; Push old mask, set new mask bits
                                       ; if P1 false, go to ELSE1
    ld.global.f64   RD2, [Y+R8]  ; RD2 = Y[i]
    sub.f64    RD0, RD0, RD2           ; Difference in RD0
    st.global.f64   [X+R8], RD0  ; X[i] = RD0
    @P1, bra ENDIF1, *Comp         ; complement mask bits
                                       ; if P1 true, go to ENDIF1
ELSE1:      ld.global.f64 RD0, [Z+R8]     ; RD0 = Z[i]
            st.global.f64 [X+R8], RD0     ; X[i] = RD0
ENDIF1:     <next instruction>, *Pop; pop to restore old mask
```

NVIDIA GPU Memory Structures

- Each SIMD Lane has private section of off-chip DRAM
  - "Private memory"
  - Contains stack frame, spilling registers, and private variables
- Each multithreaded SIMD processor also has local memory
  - Shared by SIMD lanes / threads within a block
- Memory shared by SIMD processors is GPU Memory
  - Host can read and write GPU memory

NVIDIA GPU Memory Structures

- GPU memory is shared by all Grids (vectorized loops), local memory is shared by all threads of SIMD instructions within a Thread Block (body of a vectorized loop), and private memory is private to a single CUDA Thread.
- Pascal allows preemption of a Grid, which requires that all local and private memory be able to be saved in and restored from global memory.



Pascal Architecture Innovations

- Each SIMD processor has
  - Two or four SIMD thread schedulers, two instruction dispatch units
  - 16 SIMD lanes (SIMD width=32, chime=2 cycles), 16 load-store units, 4 special function units
  - Two threads of SIMD instructions are scheduled every two clock cycles
- Four main innovations
  - of Pascal:Fast single-, double-, and half-precision fp arithmetic
    - Single precision fp of the GPU runs at a peak of 10 TeraFLOP/s.
    - Double-precision is roughly half-speed at 5 TeraFLOP/s,
    - half-precision is about double-speed at 20 TeraFLOP/s when expressed as 2-element vectors
  - High Bandwith Memory (HBM2)
    - wide bus (4096 data wires running at 0.7 GHz, peak bandwidth of 732 GB/s)
  - High-speed chip-to-chip interconnect
    - NVLink between multiple GPUs (20 GB/s in each direction)
  - Unified virtual memory and paging support

Pascal Multithreaded SIMD Proc.

Vector Architectures vs GPUs

- Both architectures are designed to execute data-level parallel programs
- Multiple SIMD Processors in GPUs act as independent MIMD cores, just as many vector computers have multiple vector processors
- Multithreading is fundamental to GPUs, but missing from most vector processors
- Registers
  - RV64V register file holds entire vectors, GPU distributes vectors across the registers of SIMD lanes
  - RV64 has 32 vector registers of 32 elements (1024), GPU has 256 registers with 32 elements each (8192), supporting multithreading
  - RV64 has 2 to 8 lanes with vector length of 32, chime is 4 to 16 cycles, a multithreaded SIMD processor chime is 2 to 4 cycles
- The closest GPU term to a vectorized loop is Grid

SIMD Architectures vs GPUs

- All GPU loads are gather instructions and all GPU stores are scatter instructions
- GPUs have more SIMD lanes
- GPUs have hardware support for more threads
- Both have 2:1 ratio between double- and single-precision performance
- Both have 64-bit addresses, but GPUs have smaller memory
- SIMD architectures have no scatter-gather support

SIMD Architectures vs GPUs

- Similarities and differences between multicore with multimedia SIMD extensions and recent GPUs

Feature	Multicore with SIMD	GPU
SIMD Processors	4–8	8–32
SIMD Lanes/Processor	2–4	up to 64
Multithreading hardware support for SIMD Threads	2–4	up to 64
Typical ratio of single-precision to double-precision performance	2:1	2:1
Largest cache size	40 MB	4 MB
Size of memory address	64-bit	64-bit
Size of main memory	up to 1024 GB	up to 24 GB
Memory protection at level of page	Yes	Yes
Demand paging	Yes	Yes
Integrated scalar processor/SIMD Processor	Yes	No
Cache coherent	Yes	Yes on some systems

Loop-Level Parallelism

- Loops in programs are the fountainhead of many of the types of parallelism
- Finding and manipulating loop-level parallelism is critical to exploiting both DLP and TLP, as well as the more aggressive static ILP approaches
- Loop-level parallelism is investigated at the source level or close to it,
  - while most analysis of ILP is done once instructions have been generated by the compiler.
- Loop-level analysis involves determining what dependences exist among the operands in a loop across the iterations of that loop.
  - Data dependences arise when an operand is written at some point and read at a later point.
  - Name dependences also exist and may be removed by the renaming techniques

Loop-Level Parallelism

- Analysis of loop-level parallelism focuses on determining whether data accesses in later iterations are dependent on data values produced in earlier iterations;
  - such dependence is called a loop-carried dependence.
- **Example 1**

```
for (i=999; i>=0; i=i-1)
    x[i] = x[i] + s;
```

  - Two uses of x[i] are dependent,
    - this dependence is within a single iteration and is not loop-carried.
  - There is a loop-carried dependence between successive uses of i in different iterations,
    - this dependence involves an induction variable that can be easily recognized and eliminated.

Loop-Level Parallelism

- **Example 2**
- Consider a loop like this one:

```
for (i=0; i<100; i=i+1) {
    A[i+1] = A[i] + C[i];   /* S1 */
    B[i+1] = B[i] + A[i+1];    /* S2 */
}
```

- What are the data dependences among the statements S1 and S2 in the loop?

Loop-Level Parallelism

- **Answer:**

```
for (i=0; i<100; i=i+1) {
      A[i+1] = A[i] + C[i];   /* S1 */
      B[i+1] = B[i] + A[i+1];     /* S2 */
}
```

- There are two different dependences:
  - S1 uses a value computed by S1 in an earlier iteration,
    - because iteration i computes A[i+1], which is read in iteration i+1.
    - The same is true of S2 for B[i] and B[i+1].
  - S2 uses the value A[i+1] computed by S1 in the same iteration

Loop-Level Parallelism

- These two dependences are distinct and have different effects.
- Assuming that only one of these dependences exists at a time,
  - because the dependence of statement S1 is on an earlier iteration of S1, this dependence is loop-carried.
  - This dependence forces successive iterations of this loop to execute in series.
- 2nd dependence is within an iteration and is not loop-carried.
  - Thus, if this were the only dependence, multiple iterations of the loop would execute in parallel,
    - as long as each pair of statements in an iteration were kept in order.

Loop-Level Parallelism

- **Example 3**
- Consider a loop like this one:

```
for (i=0; i<100; i=i+1) {
    A[i] = A[i] + B[i];      /* S1 */
    B[i+1] = C[i] + D[i];    /* S2 */
}
```

- What are the dependences between S1 and S2?
- Is this loop parallel?
- If not, show how to make it parallel.

Loop-Level Parallelism

- **Answer**

```
for (i=0; i<100; i=i+1) {
        A[i] = A[i] + B[i];      /* S1 */
        B[i+1] = C[i] + D[i];    /* S2 */
}
```

- S1 uses value computed by S2 in previous iteration
  - so there is a loop-carried dependence between S2 and S1
- But dependence is not circular so loop is parallel
  - A loop is parallel if it can be written without a cycle in the dependences because the absence of a cycle means that the dependences give a partial ordering on the statements.

Loop-Level Parallelism

- Although there are no circular dependences in the preceding loop, it must be transformed to conform to the partial ordering and expose the parallelism
  - There is no dependence from S1 to S2
  - On the first iteration of the loop, statement S2 depends on the value of B[0] computed prior to initiating the loop
- These two observations allow us to replace the preceding loop with the following code sequence:

```
A[0] = A[0] + B[0];
for (i=0; i<99; i=i+1) {
      B[i+1] = C[i] + D[i];
      A[i+1] = A[i+1] + B[i+1];
}
B[100] = C[99] + D[99];
```

- The dependence between the two statements is no longer loop-carried so that iterations of the loop may be overlapped, provided the statements in each iteration are kept in order.

Loop-Level Parallelism

- **Example 4**
- Consider the following code:

```
for (i=0;i<100;i=i+1)  {
        A[i] = B[i] + C[i];
        D[i] = A[i] * E[i];
}
```

- The second reference to A in this example need not be translated to a load instruction because we know that the value is computed and stored by the previous statement.
  - Thus the second reference to A can simply be a reference to the register into which A was computed.

Loop-Level Parallelism

- **Example 5**
- Often loop-carried dependences are in the form of a recurrence.
- A recurrence occurs when a variable is defined based on the value of that variable in an earlier iteration, usually the one immediately preceding,
- Consider the following code:

```
for (i=1;i<100;i=i+1)  {
      Y[i] = Y[i-1] + Y[i];
}
```

- Detecting a recurrence can be important for two reasons:
  - some architectures (especially vector computers) have special support for executing recurrences,
  - in an ILP context, it may still be possible to exploit a fair amount of parallelism.

Finding dependencies

- Finding the dependences in a program is important
  - to determine which loops might contain parallelism
  - to eliminate name dependences.
- The complexity of dependence analysis arises also because of the presence of
  - arrays and pointers in languages such as C or C++,
  - pass-by-reference parameter passing in Fortran
- How does the compiler detect dependences in general?
  - Nearly all dependence analysis algorithms work on the assumption that array indices are affine

Finding dependencies

- A one-dimensional array index is affine if it can be written in the form:
  - $a \times i + b$ (a and b are constants and i is loop index)
- Index of a multidimensional array is affine if the index in each dimension is affine
- Assume:
  - Stored to $a \times i + b$, then
  - Loaded from $c \times i + d$
    - where i is the for-loop index variable that runs from m to n
  - Dependence exists if:
    - Given j, k such that $m \leq j \leq n$, $m \leq k \leq n$
    - Store to $a \times j + b$, load from $a \times k + d$, and $a \times j + b = c \times k + d$

Finding dependencies

- Generally cannot be determined at compile time
- A simple and sufficient test for the absence of a dependence is the greatest common divisor (GCD) test:
  - It is based on the observation that if a loop-carried dependence exists, then GCD(c, a) must divide ($d - b$).
    - Recall that an integer, x, divides another integer, y, if we get an integer quotient when we do the division y/x and there is no remainder.

Finding dependencies

- **Example:**
- Use the GCD test to determine whether dependences exist in the following loop:

```
for (i=0; i<100; i=i+1) {
X[2*i+3] = X[2*i] * 5.0;
}
```

- **Answer**
- Given the values $a = 2$, $b = 3$, $c = 2$, and $d = 0$, then GCD(a, c) = 2, and $d - b = -3$.
- Because 2 does not divide -3, no dependence is possible.

Finding dependencies

- The GCD test is sufficient to guarantee that no dependence exists;
  - however, there are cases where the GCD test succeeds but no dependence exists.
    - This can arise, for example, because the GCD test does not consider the loop bounds.
- In general, determining whether a dependence actually exists is NP-complete.
- In addition to detecting the presence of a dependence, a compiler wants to classify the type of dependence.
  - This classification allows a compiler to recognize name dependences and eliminate them at compile time by renaming and copying.

Finding dependencies

- **Example**
- The following loop has multiple types of dependences.
- Find all the true dependences, output dependences, and antidependences, and eliminate the output dependences and antidependences by renaming.

```
for (i=0; i<100; i=i+1) {
    Y[i] = X[i]  / c;          /* S1 */
    X[i] = X[i] + c;      /* S2 */
    Z[i] = Y[i] + c;           /* S3 */
    Y[i] = c - Y[i];      /* S4 */
}
```

Finding dependencies

- **Answer**
- The following dependences exist among the four statements:
  - There are true dependences from S1 to S3 and from S1 to S4 because of Y[i].
    - These are not loop-carried, so they do not prevent the loop from being considered parallel.
    - These dependences will force S3 and S4 to wait for S1 to complete.
  - There is an antidependence from S1 to S2, based on X[i].
  - There is an antidependence from S3 to S4 for Y[i].
  - There is an output dependence from S1 to S4, based on Y[i].

Finding dependencies

- The following version of the loop eliminates these false (or pseudo) dependences.

```
for (i=0; i<100; i=i+1 {
 T[i] = X[i] / c;       /* Y renamed to T to remove output
                           dependence */
 X1[i] = X[i] + c;    /* X renamed to X1 to remove
                           antidependence */
 Z[i] = T[i] + c;       /* Y renamed to T to remove
                           antidependence */
 Y[i] = c - T[i];
}
```

Eliminating Dependent Computations

- One of the most important forms of dependent computations is a recurrence.
  - A dot product is a perfect example of a recurrence:

```
for (i=9999; i>=0; i=i-1)
        sum = sum + x[i] * y[i];
```

- This loop is not parallel
  - because it has a loop-carried dependence on the variable sum
- Transform to…

```
for (i=9999; i>=0; i=i-1)
        sum [i] = x[i] * y[i];
for (i=9999; i>=0; i=i-1)
        finalsum = finalsum + sum[i];
```

- In 1st loop, sum has been expanded from a scalar into a vector quantity
  - This transformation is called scalar expansion
    - makes this new loop completely parallel.
- 2nd loop is the reduce step
  - Although this loop is not parallel, it has a very specific structure called a reduction.
  - Reductions are common in linear algebra

Eliminating Dependent Computations

- Reductions are sometimes handled by special hardware in a vector and SIMD architecture that allows the reduce step to be done much faster than it could be done in scalar mode.
  - These work by implementing a technique similar to what can be done in a multiprocessor environment.
- Suppose for simplicity we have 10 processors.
  - In the first step of reducing the sum, each processor executes the following (with p as the processor number ranging from 0 to 9):

```
for (i=999; i>=0; i=i-1)
      finalsum[p] = finalsum[p] + sum[i+1000*p];
```

- This loop is completely parallel.
  - A simple scalar loop can then complete the summation of the last 10 sums.
- Similar approaches are used in vector processors and SIMD Processors.
- It is important to observe that the preceding transformation relies on associativity of addition.

Fallacies and Pitfalls

- GPUs suffer from being coprocessors
  - GPUs have flexibility to change ISA
- Concentrating on peak performance in vector architectures and ignoring start-up overhead
  - Overheads require long vector lengths to achieve speedup
- Increasing vector performance without comparable increases in scalar performance
- You can get good vector performance without providing memory bandwidth
- On GPUs, just add more threads if you don't have enough memory performance